new_theory ‘HOL’;;

An Introduction to
Hardware Verification
in
Higher Order Logic

by

Graham Birtwistle, Computer Science, University of Calgary,
Net address: graham@cpsc.ucalgary.ca

Shiu-Kai Chin, Computer Engineering, Syracuse University,
Net address: chin@cat.syr.edu

Brian Graham, Computing Laboratory, Cambridge University,
Net address: btg@cl.cam.ac.uk

DRAFT 13/08/94

## Contents

Apologia xiii

Preface xv

### Part I ML

1 Programming in ML 3
   1.1 Getting started 4
   1.2 Basic ML 4
      1.2.1 Functions, pairs and lists 9
   1.3 Typing and polymorphism 19
   1.4 Higher order functions 21
   1.5 λ expressions 23
   1.6 Failure 25
   1.7 ML datatypes 26
   1.8 Abstract datatypes 30
      1.8.1 Complex numbers 30
      1.8.2 Integers: I 33
      1.8.3 Integers: II 36
      1.8.4 Structural induction 38
Example: A mapping theorem 40
Exercises 1 41

2 Terms in \(p\)HOL 43
   2.1 \(p\)HOL—the propositional subset of HOL 43
   2.2 A datatype for \(p\)HOL terms 44
   2.3 Lexical analysis of \(p\)HOL 48
   2.4 Parsing \(p\)HOL 50
Exercises 2 54
## CONTENTS

### Part I  HOL

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>A formal system for $\mathcal{H}$HOL</td>
<td>55</td>
</tr>
<tr>
<td>3.2</td>
<td>Doing proofs in $\mathcal{H}$HOL</td>
<td>62</td>
</tr>
<tr>
<td>3.3</td>
<td>A datatype for $\mathcal{H}$HOL theorems</td>
<td>63</td>
</tr>
<tr>
<td>3.4</td>
<td>Machine checked proofs in $\mathcal{H}$HOL</td>
<td>69</td>
</tr>
<tr>
<td>3.5</td>
<td>Summary</td>
<td>72</td>
</tr>
</tbody>
</table>

### Exercises 3 | 73

### Part II  HOL

### 4 The HOL notation

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>The HOL first order subset</td>
<td>77</td>
</tr>
<tr>
<td>4.2</td>
<td>Higher order logic</td>
<td>79</td>
</tr>
<tr>
<td>4.3</td>
<td>Types</td>
<td>82</td>
</tr>
<tr>
<td>4.4</td>
<td>Other background material</td>
<td>85</td>
</tr>
<tr>
<td>4.5</td>
<td>Formal proofs with the HOL system</td>
<td>87</td>
</tr>
<tr>
<td>4.6</td>
<td>Rules for quantifiers</td>
<td>90</td>
</tr>
<tr>
<td>4.7</td>
<td>Backward proof</td>
<td>97</td>
</tr>
</tbody>
</table>

### Exercises 4 | 97

### 5 Verifying hardware

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Combinational circuits</td>
<td>101</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Specification</td>
<td>104</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Defining implementations</td>
<td>112</td>
</tr>
<tr>
<td>5.2</td>
<td>Verification of circuits</td>
<td>117</td>
</tr>
</tbody>
</table>

### Exercises 5 | 122

### 6 Uses and limitations of verification

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Hierarchies</td>
<td>123</td>
</tr>
<tr>
<td>6.2</td>
<td>Simulation</td>
<td>124</td>
</tr>
<tr>
<td>6.3</td>
<td>Simulation vs formal verification</td>
<td>126</td>
</tr>
<tr>
<td>6.4</td>
<td>Limitations to verification</td>
<td>127</td>
</tr>
</tbody>
</table>
## CONTENTS

### Part III  Starting with the HOL system

**7  A basic library of gates**

7.1  A theory of basic gates .......................... 135
7.2  Verification of the mux2 .......................... 137
    7.2.1  Proof structure ............................ 137
    7.2.2  Verification ............................... 138
Exercises 7 ........................................ 145

**8  Bits, numbers, and words**

8.1  Techniques I—arithmetic constants ............... 147
    8.1.1  Conversion to standard form ............... 148
    8.1.2  Equality of arithmetic constants .......... 148
    8.1.3  Less than comparisons .................... 150
    8.1.4  Other comparisons ........................ 151
8.2  Bits—useful facts about booleans ................ 152
8.3  Techniques II—the assumption list ............... 154
8.4  Nums—useful facts about numbers ................. 156
8.5  Techniques III—induction ....................... 161
8.6  Words—values on busses ......................... 163
Exercises 8 ........................................ 170

**9  Comparators**

9.1  A simple 1-bit comparator ........................ 171
    9.1.1  Specification .............................. 171
    9.1.2  Consequences of the specification ........ 172
9.2  Techniques V—Theorems on the fly ............... 175
    9.2.1  Implementation ............................ 178
    9.2.2  Verification .............................. 179
9.3  bitComp ........................................ 182
    9.3.1  Specification .............................. 184
    9.3.2  Consequences of the specification ........ 184
    9.3.3  Implementation ............................ 189
    9.3.4  Verification .............................. 190
9.4  Word comparator ................................. 193
    9.4.1  Specification .............................. 193
    9.4.2  Consequences of the specification ........ 194
9.4.3 Implementation ........................................ 195
9.4.4 Verification ............................................. 196
9.5 Techniques VI — val and $2^n$ ............................ 201

Part IV ALU case study ........................................ 207

10 Step 1: ALU overview ...................................... 209
   Exercises 10 .................................................. 217

11 Step 2: The adder sub-system ................................ 219
   11.1 Techniques V: theorem continuations ................ 219
   11.2 Techniques VI: dealing with subtraction ............. 222
   11.3 Techniques VII: disjunctive case splits .............. 224
   11.4 Verification of nAdder .................................. 226
      11.4.1 nAdder specification: test I ...................... 227
      11.4.2 nAdder specification: test II ..................... 230
      11.4.3 Verification of the adder sub-system ............ 236
   Exercises 11 .................................................. 248

12 Step 3: ALU verification .................................... 253
   12.1 The nAddXor box ....................................... 254
      12.1.1 Verification of AX ................................. 254
      12.1.2 Verification of nAX ............................... 256
      12.1.3 Verifying nAddXor ............................... 261
   12.2 The nMod box ........................................... 264
      12.2.1 Verification of the 1-bit Mod .................... 265
      12.2.2 Verification of nMod ............................. 267
   12.3 ALU verification ....................................... 269
      12.3.1 Verifying the arithmetic operations ............... 272
      12.3.2 Verifying the logical operations ................. 277
   Exercises 12 .................................................. 288
Part V  Filling the HOLes

13 Subst., conv. and rewriting  291
   13.1 Representing HOL terms  293
      13.1.1 Primitive HOL terms  293
      13.1.2 Non-primitive HOL terms  296
      13.1.3 Substitution in terms  298
      13.1.4 Substitution for free terms  301
      13.1.5 Simultaneous substitution  302
      13.1.6 subst  302
   13.2 Representing HOL theorems  304
      13.2.1 Substitution in theorems  304
   13.3 Conversions  306
      13.3.1 Converting subterms of terms  308
      13.3.2 Combining forms for conversions  309
      13.3.3 Depth conversions  311
      13.3.4 Tools for hardware verification  313
   13.4 Forward inference  317
      13.4.1 Primitive inference rules  317
      13.4.2 Derived inference rules  319
   13.5 Rewriting  321
   Exercises 13  327

14 Tactics and tacticals  329
   14.1 Tactics  329
      14.1.1 Tactics related to the structure of the goal  329
      14.1.2 Tactics for case analysis on the goal  332
      14.1.3 Tactics for manipulating assumptions  333
      14.1.4 Tactics to use theorems (and assumptions)  340
      14.1.5 Tactics for substitution  343
      14.1.6 Tactics for rewriting  345
      14.1.7 Miscellaneous tactics  346
   14.2 Tacticals  346
      14.2.1 List tacticals  347
CONTENTS

15 Theorem continuations 349
  15.1 Theorem Generators ........................................... 349
    15.1.1 DISCH\_THEN: working with implications ........... 349
    15.1.2 Resolvents ............................................... 352
  15.2 Manipulating Theorems ......................................... 359
    15.2.1 Existential quantifiers ................................. 359
    15.2.2 Conjunctions ............................................ 360
    15.2.3 Disjunctions ............................................. 362
    15.2.4 Combinations and Permutations ....................... 364
  15.3 General Induction ............................................ 364
  15.4 Proof Hacking ................................................. 365

Part VI  Clocked sequential circuits 367

16 Basic techniques 369
  16.1 Library circuits ............................................... 371
  16.2 Non-primitive circuits ....................................... 375
    16.2.1 Definition of Reg ...................................... 375
    16.2.2 Defining nReg .......................................... 378
    16.2.3 Verification of Reg ..................................... 380
    16.2.4 Verification of nReg ................................... 384
    16.2.5 Line of delays .......................................... 391
  Exercises 16 ....................................................... 397

17 Case study I: shifters 399
  17.1 Shifter ......................................................... 399
    17.1.1 Verification of Shift ................................... 399
    17.1.2 Verification of nShift ................................ 401
  17.2 Staged shifter ................................................ 413
    17.2.1 Modulo arithmetic ...................................... 413
    17.2.2 nMux21 .................................................. 416
    17.2.3 Verification of nbs .................................... 417
    17.2.4 Verification of nSS .................................... 419
A HOL built-ins
   A.1 Axioms ........................................... 579
   A.2 Primitive inference rules ....................... 579
   A.3 Manipulating terms ............................... 580
   A.4 Manipulating theorems ........................... 582
   A.5 Basic rewrite rules .............................. 582

B Extras ............................................ 585
   B.1 ML code ........................................ 585

C Theories developed for this text ................. 587
   C.1 Pairs.ml .......................................... 587
   C.2 Boos.ml ........................................ 589
   C.3 bits.ml ........................................ 590
   C.4 nums.ml .......................................... 593
   C.5 words.ml ......................................... 598
## List of Figures

1.1 Standard ML environment .......................... 5
1.2 Augmented environment .......................... 6
1.3 Effect of let definitions .......................... 6
1.4 Evaluation of let \( x = a + b \) in \( a \times x \) .......................... 7
1.5 let and letrec bindings .......................... 14

3.1 Proof tree for \( \vdash d \) .......................... 56

4.1 Line of unit delays .......................... 81

5.1 Typical hardware device .......................... 104
5.2 The mux2 gate .......................... 105
5.3 The full adder .......................... 106
5.4 nAdder \( n \) .......................... 108
5.5 Buss with \( n \) wires .......................... 109
5.6 mux2 implementation .......................... 112
5.7 Equivalent mux2 implementation .......................... 113
5.8 Another equivalent mux2 implementation .......................... 113
5.9 n-bit adder implementation .......................... 115
5.10 Recursive construction of an \( n+1 \)-bit adder .......................... 115
5.11 nInv implementation .......................... 119

6.1 4-bit adder: nAdder 4 a b cin s cout .......................... 124

9.1 Comp .......................... 171
9.2 Comp implementation .......................... 179
9.3 Word comparator .......................... 183
9.4 bitComp .......................... 184
9.5 nComp .......................... 193
9.6 Word comparator .......................... 195

10.1 ALU .......................... 209
10.2 AX: the modified full adder .......................... 211
<table>
<thead>
<tr>
<th>FIGURE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.3</td>
<td>ALU: refinement 1</td>
</tr>
<tr>
<td>10.4</td>
<td>Modifying the b inputs</td>
</tr>
<tr>
<td>10.5</td>
<td>Modifying the a inputs</td>
</tr>
<tr>
<td>12.1</td>
<td>ALU proof hierarchy</td>
</tr>
<tr>
<td>12.2</td>
<td>The nAddXor</td>
</tr>
<tr>
<td>12.3</td>
<td>Mod box</td>
</tr>
<tr>
<td>16.1</td>
<td>Sequential sub-net</td>
</tr>
<tr>
<td>16.2</td>
<td>1-bit register</td>
</tr>
<tr>
<td>16.3</td>
<td>1-bit register</td>
</tr>
<tr>
<td>16.4</td>
<td>1-bit register</td>
</tr>
<tr>
<td>16.5</td>
<td>1-bit register implementation</td>
</tr>
<tr>
<td>16.6</td>
<td>n-bit register</td>
</tr>
<tr>
<td>16.7</td>
<td>nReg implementation</td>
</tr>
<tr>
<td>16.8</td>
<td>Delay by n wiring and implementation</td>
</tr>
<tr>
<td>17.1</td>
<td>1-bit shift register</td>
</tr>
<tr>
<td>17.2</td>
<td>n × m staged shifter</td>
</tr>
<tr>
<td>17.3</td>
<td>n × m staged shifter implementation</td>
</tr>
<tr>
<td>18.1</td>
<td>1-bit up counter</td>
</tr>
<tr>
<td>21.1</td>
<td>Relating Specification and Implementation Behaviours</td>
</tr>
<tr>
<td>21.2</td>
<td>Relating Specification and Implementation Output Functions</td>
</tr>
<tr>
<td>21.3</td>
<td>eval_O function</td>
</tr>
<tr>
<td>21.4</td>
<td>Defining the Machine behaviour</td>
</tr>
<tr>
<td>21.5</td>
<td>Defining the machine output</td>
</tr>
<tr>
<td>21.6</td>
<td>Correspondence of outputs</td>
</tr>
</tbody>
</table>
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Representation of terms in $\mathcal{F}$HOL</td>
<td>44</td>
</tr>
<tr>
<td>2.2</td>
<td>BNF for $\mathcal{F}$HOL</td>
<td>51</td>
</tr>
<tr>
<td>4.1</td>
<td>Operator precedences</td>
<td>78</td>
</tr>
<tr>
<td>4.2</td>
<td>Primitive recursion</td>
<td>86</td>
</tr>
<tr>
<td>5.1</td>
<td>Multiplexor truth table</td>
<td>105</td>
</tr>
<tr>
<td>5.2</td>
<td>1-bit full adder truth table</td>
<td>107</td>
</tr>
<tr>
<td>5.3</td>
<td>2-bit adder truth table</td>
<td>109</td>
</tr>
<tr>
<td>6.1</td>
<td>Formal and simulation specifications of the 1-bit full adder</td>
<td>126</td>
</tr>
<tr>
<td>6.2</td>
<td>Formal and simulation specifications of the word adder</td>
<td>127</td>
</tr>
<tr>
<td>10.1</td>
<td>ALU control bit encodings</td>
<td>210</td>
</tr>
</tbody>
</table>
Apologia

Work started on this book in 1986 and progressed slowly as we stumbled towards its current style of presentation and content. Our intent was to have it published as a formal text but we have been overtaken by two major events.

The first event we choose to gloss over, except to say that it pays to choose a supportive home base. The second event was the advent of the SML-based HOL90. The text should be rewritten from end-to-end but given the aggravation and disruption stemming from event one, we have neither the time nor the energy to do that. However we do retain copyright to the material and might get round to it in the future.

What to do with the 600 pages? We think it best to make the text available to the HOL community as is, warts and all. Besides a final polishing (the accounts of polymorphism for ML and HOL could be combined, the base case for induction for arithmetic circuits should perhaps be wires rather than a circuit), there are some embarrassing omissions (chiefly an index, chapter summaries, advice on how to use the help facility, tautology checking and other advances in mechanisation, and updated references). For these we apologise in advance.

Finally we would like to thank the staff at Prentice Hall who have patiently expected us to deliver this text for some time and graciously accepted our reasons for giving up. The text went through two reviews by their technical reviewers who were very perceptive and constructive. We owe them a great deal and are sorry to let them down.
The specification and verification work in this text is carried out in the HOL (Higher Order Logic) proof assistant, designed and written by Mike Gordon at the University of Cambridge [46]. HOL implements a version of Church's typed higher order logic [1, 23]. We work with HOL because a higher order logic permits clear, readable, and succinct specifications. Its higher order features are particularly useful when it comes to defining abstractions, specifying architectures, and joining subsystems together. The disadvantage is that with increased expressive power, less can be automated. In fact, the HOL system is a proof-checker (or proof-assistant) and not an automated theorem prover. You have to supply to HOL the detailed working of what you think is a proof and HOL checks your proof step by step.

The book is based upon experiences with a graduate level Computer Science course in Hardware Verification given at the University of Calgary since 1985. It is aimed at graduate and senior undergraduate students in Computer Engineering and Computer Science and at mature readers requiring a self-study introduction to verifying hardware using HOL. The text assumes some prior knowledge of first order logic, functional programming and the lambda calculus, primitive recursion, and digital design. These topics are covered by most undergraduate curricula and in many excellent text books. On the other hand, we anticipate that the arts of hardware specification and verification and the HOL notation will be new to most readers. We aim to give you a good feel for these topics before you attempt your first session with the HOL system. This is deliberate. We feel that once you are at ease with the basic ideas and principles, you can focus much more sharply on becoming adept at using the HOL system.

In the main we have adopted an informal approach and present information on the need to know basis; i.e. we tell you just enough to tackle the current problem, rather than going systematically through all the features of HOL. The examples in the text start at the gate level and increase in complexity up to the sub-system level (arithmetic units, registers, shifters, stacks). As a rider to this approach, there is some unevenness in our development of HOL methods and tools. Accordingly, in chapters 8, 13-15 we take time out from verification examples to survey and catalogue some common HOL tools and their uses and explain how they fit together.

Chapters 1–3 introduce the most useful features of ML [59, 104] the language in which HOL is hosted, and uses them to write a simple proof
checker for propositional logic. Chapter 1 introduces the basics of programming in ML using the built-in pair and list data types before moving on to the use of higher order functions, user-definable data types, and abstract data types. Chapters 2 and 3 implement a proof checker for the propositional logic subset of HOL. Chapter 2 reminds you of propositional logic, defines an abstract data type for terms (valid propositional logic expressions), and then describes a lexer and parser for terms. Chapter 3 gives an axiomatisation of propositional logic (a selected set of axioms and inference rules) and shows you how to carry out proofs in this system. We then implement this axiomatisation using an abstract data type for theorems and show it in action with several more proofs. At the end of these chapters, you should be competent in ML, have a basic understanding of how to implement a proof checker, and be able to carry out proofs in a formal manner. The extended example in chapters 2 and 3 serve as an introduction to mechanical proof checking and will give you a good feel for how the HOL system is implemented.

Chapters 4–6 lay part of the foundation for formal work with the HOL proof assistant. Through examples, we teach you the HOL notation, how to specify hardware in HOL, how to conduct proofs in HOL, before giving you an understanding of the possibilities and limitations of the verification methodology. Having completed these chapters, you should be ready to get acquainted with the HOL proof assistant itself. Chapter 4 reminds you of the basic properties of first order logic before sketching the lambda calculus and some of the notation and intuition of higher order logic. It also serves to remind you of things we expect you to know, especially primitive recursion and induction which are used to specify and verify regular VLSI structures. In chapter 4 we also carry out two important forward proofs and introduce the notion of backward proof. In chapter 5, we show you how to specify some simple circuits and regular sub-systems using the HOL notation and stress the need for general solutions that can be applied to families of designs. We also show you how to define implementations of circuits and sub-systems. Finally, we do some paper and pencil verifications of hardware designs. In chapter 6 we show how verification fits into the world of VLSI CAD and point out some of its inherent limitations.

By now you should feel comfortable with using the ML subcomponent of the HOL system, with paper and pencil descriptions of hardware in HOL, and with paper and pencil proof outlines. Now is the time to get you started with the HOL system, which is the purpose of chapters 7–9. The initial hurdle we face is this: you will not be able to complete non-trivial proofs with HOL until you know HOL well, and you will not know HOL well until you have completed several proofs. Our first session with the HOL system
in chapter 7 merely shows you how to make suitable definitions of some hardware primitives and save them in a theory. We then verify a two input multiplexer built from two inverters and three nand gates. This example shows you how to bring down and use the results saved in a theory and how to extend a theory. We then ask you to verify a number of commonly used gates and save them in a library. The gates include two-, three-, four-, and five-input and and or gates, and the four-input multiplexer. We then devote chapter 8 to building three more useful libraries of facts about bits, numbers, and words which are required for our next hardware proofs. The examples in the text illustrate the value of standard forms, show you how to make primitive recursive definitions and carry out proofs by induction. In chapter 9 we specify and verify both a 1-bit comparator and a ripple comparator subsystem. We also show you how to develop, prove and apply your own mini-theorems on-the-fly whilst working on a larger proof. The proofs in these chapters are completed using the backward proof strategy. Using this strategy, one first sets up the goal to be proved, and then uses HOL tactics to split the goal into simpler parts. If any sub-goal is itself tricky, then we apply the same strategy repeatedly. Eventually we wind up with several small sub-goals each of which is trivial to prove. There will be many ways in which a particular verification can be carried out. In general we eschew tricks, and try to present a straightforward, general and robust style of theorem proving.

Chapters 10–12 form a substantial case study which develops the specification and verification of an ALU from scratch. In these chapters, we stress specification and proof development and do not expound all the details of all the proofs. These proofs show you how to exploit hierarchy in proof development and thus take advantage of completed work instead of flattening every proof down to the most primitive level (a large network of inverters, nands, and nors).

Since our development so far has been driven by examples, there are some holes in our knowledge of HOL. Chapters 13–15 systematically work through the whole catalogue of HOL tools and methods. We develop our presentation working upwards. In chapter 13, we start at the bottom of the hierarchy. We first explain how terms are represented and they may be constructed and taken apart. Then we define the mathematics of substitution on terms and present the HOL primitives for substitution. We then cover conversions, which take terms to theorems. A conversion only applies to a term at the top level. Some control structure is required to navigate through a term and apply a conversion to its sub-terms, once or repeatedly. HOL has several defined primitives which can be combined to produce these effects; analogous to tactics, they are called conversionals.
Finally we survey and catalogue the different styles of forward inferencing. In chapter 14 we survey and catalogue all the tactics we have used (and will use during the course of this text) and the common tacticals that can be used to weave them together. We also introduce theorem continuations which allow us to pick up and manipulate specified terms and/or theorems through several operations before letting them go. They are used by experts to good effect when they do not wish to clutter up the assumption list with assumptions that are used but once.

Chapters 16–18 are concerned with the verification of sequential circuits. Correctness proofs for sequential devices are usually more difficult than those of combinational circuits because we have to match signals that vary with time, but we can make substantial strides with a few standard techniques. Examples will include registers, stacks, counters, ...

One of our reasons for choosing to work with HOL is its generality. It is good for reasoning about algorithms and software as well as hardware. In chapters 19–21 we introduce recursive data types in HOL and show them in action by proving a few meta theorems about lists and trees. As larger examples we specify and verify insert sort and the correctness of the translation schema for a very small language running on a very small machine. These should give you a feel for the possibilities for software verification in HOL.

Acknowledgements.

This book was prepared using the \LaTeX document preparation system [69]. \LaTeX itself was built upon Knuth's \TeX [68]. The margins of the text have been adjusted to be large enough to contain any neat proofs you may stumble upon whilst using it.

The inclusion of great numbers of interactive HOL scripts demanded careful attention to their accuracy and currency. This task was aided considerably by the use of the \mweb proof script management utilities written by Wai Wong. These utilities, along with an extension written to aid the preparation of this document, are available in the contrib library of the hol88 distribution.

The Calgary VLSI group was formed in 1983. We are indebted to John Gray for putting us on the right track (verification and hardware description languages) straight away. As soon as we became aware of his work, Mike Gordon very generously gave us first his LCF\_LSM [41] proof assistant and then copies of HOL [43] as it matured into its current state. Tom Melham was the first verifer to emerge from Calgary. He has found time for regular
visits since moving to Cambridge. He has always been willing to help us and we have benefitted greatly from his knowledge and insights. John, Mike and Tom have been sources of encouragement and inspiration for many years, and we are deeply indebted.

Many former and current associates have been a great help through the years. We express our gratitude to Dan Craigen, Inder Dhingra, Ganesh Gopalakrishnan, Jungang Han, Milan Kuchta, Paliath Narendran, Dewey Val Schorre, Konrad Slind, and Sue Stodart.

We record our grateful appreciation for their support through the years to the Alberta Microelectronics Corporation, the Canadian Microelectronics Corporation, and the Communications Research Establishment, Ottawa. Last, but not least, our work would not have been possible without the generosity and understanding of the Natural Sciences and Engineering Research Council of Canada.
Part I

ML
Chapter 1

Programming in ML

ML is a general purpose functional language [38, 40, 59, 63, 104] which has been used extensively at Cambridge University, Edinburgh University, INRIA and elsewhere for implementing theorem provers. The HOL system [105, 106, 107] is written and embedded in ML. HOL proofs are ML programs and so must conform to the syntax and semantics of ML.

There are several reasons why ML is a suitable language in which to write theorem provers. First it has a strong yet flexible type system. Typing is used at compile time to check that things are used in a manner consistent with their definition. For example, in the implementation of HOL we define a type \texttt{thm} (theorem) and the ML compiler uses its strong typing to ensure that if either an argument to a function or the result returned by a function is supposed to be of type \texttt{thm}, then it is.

Second, ML has abstract datatypes which are used to group together a datatype and operations over it. Abstract datatypes are written in such a way that direct access to the datatype constructors is not possible. Instances of the data type can be created and accessed only through specially tailored functions defined within the abstract definition. HOL implements theorems as an abstract datatype and supplies only one mechanism for generating new theorems, namely the inference rule. As an example, the inference rule \textit{modus ponens} is implemented as a binary operation over the abstract datatype \texttt{thm}. The operation is implemented as a function, \texttt{MP}, which takes two theorems, say \(\Gamma \vdash A \supset B\) and \(\Gamma \vdash A\), as arguments and from them infers a new theorem \(\Gamma \vdash B\). (The turnstile \(\vdash\) is used to denote theoremhood.) The ML compiler will check that the arguments \texttt{th1} and \texttt{th2} to a call \texttt{MP th1 th2} are typed as theorems. The body of the function implementing \texttt{MP} takes apart the two theorems and makes sure that the antecedent of \texttt{th1} matches \texttt{th2} before generating the new theorem.

As we shall see in chapters 2 and 3, theorem provers are implemented by building in a basic core of "givens" — axioms and inference rules and then deriving more powerful theorems via the inference rules. We construct more powerful inference rules in much the same way. It is essential that the core axioms and inference rules are correctly implemented, otherwise invalid theorems may be generated (for example if we forget the check on
the structure of \( th_1 \) above). Clearly, the likelihood of getting a correctly-implemented final system is greatly increased when the core is very small.

Finally, ML supports and encourages the use of functions as first-class values. Patterns of ML commands in HOL proofs can be readily combined into more powerful commands. Common higher-level commands may be named and saved for later use, thus raising the level of abstraction and understanding.

1.1 Getting started

HOL is an interactive system and sessions typically consist of a sequence of basic data and function definitions followed by requests to evaluate expressions. This promotes a distinctive programming style — short, self-contained definitions followed by applications of them. It encourages a hierarchical programming style with later definitions being constructed from previous ones rather than from scratch.

The material is presented as though we were running an interactive HOL session. When we are in session mode, we use teletype font and record both the user input and the system’s response in a framed box.

```
HOL.06 Version 2.01 (SUN/4/AKCL), built on 4/12/92

#2 + #4;;
22 : int
```

Once invoked, the HOL system displays a banner, and then prompts for a user response with a hash sign \#1. The user responds by typing in an expression ending with two semi-colons `;;` to signal the termination of the current request. (A single semi-colon is used to separate items in a list.) After carriage return has been hit, HOL evaluates the request and prints its value, together with its type.

1.2 Basic ML

This section has examples of various ML ground types, permanent and temporary bindings, and the conditional expression. The ground types are

---

1 A request longer than one line actually results in several prompts being issued, one per input line. Here and in the sequel, we follow the practice established in [63] and doctor lines of hashes down to a single hash in the scripts.
bool (true and false), int (...,-2,-1,0,1,2,...), and string (characters enclosed in single back quotes, e.g. 'Whitehall 1212').

We also introduce functions, pairs, and lists. A pair is a couple of expressions, separated by a comma and enclosed in parentheses, e.g. a place name and its grid reference, such as ('Ugglegnghy', (072, 880)) where the grid reference is itself a pair giving the northing and the easting of the named place. A list is any (non-negative) number of expressions, separated one from another by a semicolon ‘;’ and enclosed in square brackets, e.g. [ 1; 2; 3; 4 ]. The components of pairs may be of different types; but all the members of a list must have the same type.

**Permanent bindings.** As with most languages, a standard ML environment is automatically provided when an ML session starts.

![Table of bindings]

An environment contains a number of *bindings* — names and associated definitions. The standard environment contains the definitions of several useful functions for pair, list and data structure manipulation. We can augment the standard environment by new bindings of our own. Exactly how the environment is implemented is of no direct concern to us — all we need to know are the rules for adding new bindings and the look-up rules for extracting definitions from it.

We can permanently augment the environment at any time using the **let** construct. **let v = expr** adds a single new binding to the environment. For example,

```
#let a = 7;;

a = 7 : int
```

adds the binding `a = 7` to the standard environment resulting in

When an environment has several levels, we lookup an identifier at the most recent level first. If we find a match (as with `a`), we take its binding. If we don't find a match, we try the previous level (as with `mem`). If we cannot find a match at all, the identifier is undefined.

We can add several bindings to the environment at the same time using
CHAPTER 1. PROGRAMMING IN ML

Figure 1.2 Augmented environment

\[
\text{let } v_1 = expr_1 \text{ and } v_2 = expr_2 \ldots \text{ and } v_n = expr_n
\]

First the expressions \( expr_k \) are evaluated one by one in the existing environment. Suppose they evaluate to \( expr_k \) respectively. Then a new level of bindings (the \( v_k = expr_k \)) is added to the environment. Thus in

\[
\text{#let } a = 3 \text{ and } b = a + 1 \text{ and } c = true \text{ and } d = \text{"hello"};
\]

\( b \) is bound to 8 and the environment now has three levels:

Figure 1.3 Effect of let definitions

When adding several bindings at once using \texttt{let--and}, identifiers must be unique (this avoids any ambiguity of lookup).
However names can be reused from level to level, as with a which is found at Level1 and at Level2. The lookup rule accepts the most recent definition it finds. Thus the a at Level1 has been rendered inaccessible — a lookup of a will return 3.

Temporary bindings. We can construct temporary environments using the construct

\[
\text{let } v_1 = \text{expr}_1 \text{ and } v_2 = \text{expr}_2 \ldots \text{ and } v_n = \text{expr}_n \text{ in body}
\]

which uses the \(v_k\) as temporary working space whilst evaluating body. As an example (see figure 1.4), let \(x = a*a + b*b\) in \(x*x*x\) first evaluates \(a*a + b*b\) in the current environment where \(a = 3\) and \(b = 8\), then adds the binding \(x = 73\) to the environment, cubes \(x\) in the new environment, and then removes the binding \(x = 73\).

\[
\begin{array}{|c|c|}
\hline
\text{bindings} & \text{bindings} \\
\hline
\text{...} & \text{...} \\
\text{map p} & \text{map p} \\
\text{snd(a, b)} = b & \text{snd}(a, b) = b \\
\text{fst(a, b)} = a & \text{fst}(a, b) = a \\
\hline
a & 7 \\
\hline
a & 3 \\
b & 8 \\
c & \text{true} \\
d & \text{`hello'} \\
\hline
\end{array}
\]

Environment:

\begin{itemize}
\item[i] on entering the let-in
\item[ii] for evaluating \(x*x*x\)
\item[iii] for evaluating \(a*a + b*b\)
\item[iv] on leaving the let-in
\end{itemize}

Figure 1.4 Evaluation of let \(x = a*a + b*b\) in \(x*x*x\)

The binding for \(x\) exists only whilst we evaluate \(x*x*x\) and is discarded once that has been achieved. An attempt to access \(x\) now results in an error.
#let x = a+b in x*x*x;;
369017 : int

#x;;

unbound or non-assignable variable x
1 error in typing
typecheck failed

where. \textit{expr where defn} is another way of writing \textit{let defn in expr} (top-down versus bottom-up — take your pick).

#x*x*x where x = a*b;;
369017 : int

\textbf{Conditions.} ML permits $b = E_1 | E_2$ as a shorthand notation for the usual \textbf{if} $b$ \textbf{then} $E_1$ \textbf{else} $E_2$.

#if c then a else b;;
3 : int

#c => a | b;;
3 : int

We will only use the shorthand notation.

\textbf{case expressions.} ML also contains \textbf{case} expressions but we defer their introduction until section 1.7 on ML datatypes.

\textbf{Typing.} With each evaluation, ML returns not only the value of an expression but also infers and displays its \textit{type}. Typing is a powerful aid to program correctness. The type inference rules of ML detect and report inconsistencies of use. For example, attempting to add the values of $b$ (an integer) and $d$ (a string) results in an error, since the built-in operator $+$ is known to require a pair of integer operands.

#b + d;;

ill-typed phrase: (b,d)
has an instance of type \texttt{(int # string)}
which should match type \texttt{(int # int)}
1 error in typing
typecheck failed

\textbf{Built-in operators.} The following primitive operators (amongst others) are built into ML: the arithmetic operators $+, -, *, /$ (representing integer division); the comparators $=, >, <$; and the logical operators &
(representing and), or, and not. To reduce the number of parentheses in expressions, operators are given precedences. Here are the precedences for the operators we use:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>*, /</td>
<td>8</td>
</tr>
<tr>
<td>+, -</td>
<td>7</td>
</tr>
<tr>
<td>=, &lt;, &gt;</td>
<td>6</td>
</tr>
<tr>
<td>not</td>
<td>5</td>
</tr>
<tr>
<td>&amp;</td>
<td>4</td>
</tr>
<tr>
<td>or</td>
<td>3</td>
</tr>
</tbody>
</table>

Operators with higher precedence bind tighter than those with lower precedence, and operators with equal precedence are evaluated from left to right. Thus 2 + 3 * 5 evaluates to 17, and 11 + 5 = 0 evaluates to false. If in doubt, use extra parentheses as in 2 = (3 * 5).

1.2.1 Functions, pairs and lists

**Functions of one argument.** Defining and using functions are at the heart of ML. Here is a definition of the successor function in ML. Note that arguments are always called by value in ML. Comments in ML are enclosed in percent signs, and may continue over several lines.

```ml
# let suc n = n + 1;;
suc = - : (int -> int)
%
% and now examples of suc in action %
#
suc 5;;
6 : int
%suc((suc 5) + 2);;
9 : int
```

When given the function definition (which is detected by the presence of the argument n), ML stores away the new definition suc and prints the name of the defined function followed by - and its type. The symbol - is ML's way of printing a functional “value”. Functions of one argument have the arrow type: α -> β where α is the type of the argument and β is the type of the result. The type of suc is inferred by the ML system as follows.

---

2The table is not complete. There are other ML operators with greater and lesser precedences.
The operator \( + \) is built-in and is only valid when applied to a pair of integer operands. The result of the application is also an integer. \( 1 \) is an integer operand and \( n \) must also be an integer. Since \( \text{succ} \) has an integer argument and an integer result, it has the type \( \text{int} \to \text{int} \).

ML uses type information to check that function calls are consistent. Here \( \text{succ} \) is a function which expects an integer argument and returns an integer result. Thus the type of \( \text{succ} \ 4 \) is \( \text{int} \to \text{int} \).

**Functions of more than one argument.** When defining functions of several variables we have the choice of treating the arguments as a single \( n \)-tuple by writing \( \text{let } f(p_1, p_2, \ldots, p_n) = \text{expression} \) or bringing in the arguments one at a time by writing \( \text{let } f \ p_1 \ p_2 \ldots \ p_n = \text{expression} \), or indeed, of mixing these styles. This is because any function of \( n \) variables may be redefined in terms of a sequence of functions which take in one variable at a time. When functions take their arguments one at a time, they are said to be *curried*.

As a simple example, we define the operation of summing two integers in two ways. The function \( \text{plus2} \) expects a pair of arguments both integers, e.g. \( (3, 4) \), and returns an integer. The type of a pair \( (\text{arg1}, \text{arg2}) \) is written \( \text{type of arg1 \# type of arg2} \) and so the type of \( \text{plus2} \) is written \( \text{int \# int} \to \text{int} \) and the compiler rightfully complains if the arguments presented are not in pair format.

```
#let plus2 (x, y) = x + y;;
plus2 = - : ((int \# int) \to int)

#plus2 (3, 4);;
7 : int

#plus2 (3, plus2 (4, 7));;
14 : int

#plus2 4 5;;
ill-typed phrase: 4
has an instance of type int
which should match type (int \# int)
1 error in typing
typecheck failed
```

The function \( \text{add2} \) has the same effect as \( \text{plus2} \) but expects two arguments, each an integer, and returns an integer. The type of \( \text{add2} \) is \( \text{int} \to (\text{int} \to \text{int}) \) (or \( \text{int} \to \text{int} \to \text{int} \) since \( \to \) associates to the right).
We may think of `add2` as a function which accepts its single argument `x`, and returns an auxiliary function which will add the value of `x` to its single argument `y`. It is thus quite legal to apply `add2` to only a single argument. We use this property to define the increment function.

The function `inc` is defined in terms of `add2`. The first argument is fixed at 1, and the resulting function is then applied to the argument you wish to be incremented. The type of `inc` is `int -> int` since it is the result of applying `add`, which has type `int -> (int -> int)`, to 1, which is of type `int`.

You can convert curried functions into uncurried ones and vice versa using built-in functions:
Recursion. We now show a little session which assumes that we are operating with non-negative integers (0, 1, 2, 3, ...). We define three primitives isZero, succ, and pred. Notice that ML is quite happy if we present a request for evaluation as a pair (or as a list); it returns the evaluation of both (the list of) arguments. We use these facilities when testing our ML code to make more than one related request at the same time.

```
#let isZero n = (n = 0)
and succ n = (n = 0) => 0 | (n + 1);
isZero = : (int -> bool)
succ = : (int -> int)
pred = : (int -> int)

#(isZero 0, isZero 6);
(true, false) : (bool # bool)

#(pred 0, pred 5);
(0, 4) : (int # int)
```

Now we use these primitives to construct functions for addition and multiplication over non-negative integers. We start with addition and use the algorithm:

\[
\begin{align*}
    add \ 0 \ n & = n \\
    add \ m \ n & = add (m - 1) (n + 1)
\end{align*}
\]

which works in the general case by recursively calling itself keeping the sum of the arguments constant, but decrementing the first argument each time. When the first argument reaches zero, the first branch in the definition is taken and the recursion stops.

```
#let add m n = isZero m => n | add (pred m) (succ n);

unbound or non-assignable variable add
1 error in typing
typecheck failed
```

What has gone wrong? With a let, the definition is evaluated in the existing environment. In this case it fails because add is not defined. When we define a recursive function, we must use letrec.

```
#letrec add m n = isZero m => n | add (pred m) (succ n);
add = : (int -> int -> int)

#(add 0 5, add 2 7);
(5, 9) : (int # int)
```
With a letrec, the name being defined is added to the environment before the evaluation of its right hand side expression.

Some confusion may arise if we attempt the let definition of a recursive function and we have a previous binding for \( f \). First the previous definition and the new one may have incompatible types:

```plaintext
#let f m n = m*n;;
f = (-) : (* -> * -> bool)

#let f n = n=0 => 1 | n*f(n-1);;

ill-typed phrase: (n, (f (n-1)))
has an instance of type (int # (int -> bool))
which should match type (int # int)
type error in typing
```

a message which is relatively easy to decipher with practice. It is worse when the old and new definitions have compatible types as in:

```plaintext
#let f n = 0;;
f = (-) : (* -> int)

#let f n = n=0 => 1 | n*f(n-1);;
f = (-) : (int -> int)

#(f 0, f 1, f 2, f 3, f 4);;
(i, 0, 0, 0, 0) : (int # int # int # int # int)
```

Here the mistake type checks and reveals only in the unexpected test patterns. Here is a correct definition using letrec:

```plaintext
#letrec f n = n=0 => 1 | n*f(n-1);;
f = (-) : (int -> int)

#f 48;;
124139155921360726706622890473737503865214933546777600000000000 : int
```

Figure 1.5 summarises the difference between let f args = expr in body and letrec f args = expr in body definitions. In let definitions we evaluate expr in the entry environment plus args, i.e. the function under definition is not accessible. In letrec definitions we evaluate expr in the entry environment plus args plus f — the function under definition is available.

**Recursion continued.** We define multiplication by repeated addition according to the algorithm
Mutually recursive definitions are defined in the obvious style

\[
\text{letrec } v_1 = \text{expr}_1 \quad \text{and} \quad v_2 = \text{expr}_2 \ldots \quad \text{and} \quad v_n = \text{expr}_n
\]

For example here are definitions of \texttt{odd} and \texttt{even}:

\[
\begin{align*}
\texttt{#letrec even n = } & (n=0) \Rightarrow \texttt{true} \mid \texttt{odd (pred n)}; \\
& \texttt{and odd n = } (n=0) \Rightarrow \texttt{false} \mid \texttt{even (pred n)}; \\
& \texttt{even = - : (int -> bool)} \\
& \texttt{odd = - : (int -> bool)} \\
\end{align*}
\]

\[
\begin{align*}
\texttt{#(even 4, odd 7);} \\
\texttt{(true, true) : (bool # bool)}
\end{align*}
\]

**Pairs.** Pairs are represented by two expressions separated by a comma and enclosed in parentheses, e.g. \((\texttt{true}, 1)\) or \(((\texttt{true}, \texttt{false}), 1)\). The components of a pair may be of different types. Pairs of the same type may be checked for equality with \(=\).
Two operators on pairs are built-in: \texttt{fst} and \texttt{snd}, with the obvious properties that \texttt{fst(p, q) = p} and \texttt{snd(p, q) = q}. Here is some ML code to create two pairs of integers and “add” them component-wise.

```ml
let a = (3, 5) and b = (4, 17);;
\texttt{a = (3, 5)} : (int # int)
\texttt{b = (4, 17)} : (int # int)

fst a;;
\texttt{3} : int

snd b;;
\texttt{17} : int

let addPAIR p q = (fst p + fst q, snd p + snd q);;
addPAIR = - : ((int # int) -> (int # int) -> (int # int))

addPAIR a b;;
\texttt{(7, 22)} : (int # int)
```

ML accepts the n-tuple notation as shorthand for nested pairs, e.g. \((a_1, a_2, \ldots, a_n)\) may be used to represent \((a_1, (a_2, (\ldots, (a_{n-1}, a_n)))\)). The components of an n-tuple are accessed using repeated calls on \texttt{fst} and \texttt{snd} as illustrated below.

```ml
let t4 = (5, true, (1, true), 'hol');;
t4 = (5, true, (1, true), 'hol') : (int # bool # (int # bool) # string)

(fst t4, fst(snd t4), fst(snd(snd t4)), snd(snd(snd t4)));;
(5, true, (1, true), 'hol') : (int # bool # (int # bool) # string)
```

We can of course go in and access the fields of the third component in the n-tuple which is itself a pair.
Lists. ML uses the square brackets ‘[’ and ‘]’ to enclose the elements of a list and the semicolon ‘;’ to separate list elements. The elements of a list must all be of the same type. As examples, [ 3; 4; 5 ] is an integer list of length 3; [ true; false ] is a boolean list of length 2; and [ (false, 2) ] is a (bool # int) list of length 1. The empty list is denoted by []. Lists of the same type may be compared using =.

```ml
# let (l, r) = fst snd snd t4;;
l = 1 : int
r = true : bool
```

The following operations on lists are built-in:

- `mem a L` returns `true` if the atom `a` is a member of the list `L`. An error if `a` and `L` have incompatible types.
- `hd L` returns the first element in the list `L`. An error if `L` is the empty list.
- `tl L` removes the `hd` from `L` and returns the rest of the list. An error if `L` is the empty list.
- `h::L` “conses” the atom `h` onto the list `L`. An error if `h` and `L` do not have compatible types.
- `L@M` appends the lists `L` and `M`. An error if `L` and `M` do not have compatible types.
- `null L` returns `true` if `L = []`, `false` otherwise.

Here are some examples
When we define functions over numbers recursively, we do so by cases, giving separate definitions for the case zero; and for the non-zero case \((SUC \, n)\) in terms of value of the function on \(n\). Similarly when we define recursive functions over lists, we give separate cases for the empty list \([\,\,]\); and for the non-empty list \(h\cdot T\) in terms of the function on \(T\).

The simplest example is to find the length of a list. The empty list has length \(= 0\). A non-empty list with head \(h\) and tail \(T\) has length one greater than that of \(T\).

```
#let rec length L = null L => 0 | 1 + length (tl L);;
length = - : (* list -> int)
```

Our second list example appends one list to another using the cons
Append is of course built into ML (as @) but it makes a nice illustration since it contains calls on four list built-ins and defines the fifth.

```
#let rec append A B = null A => B | (hd A) . (append (tl A) B);
append = - : (* list -> * list -> * list)
#append [ 1 ; 2 ] [ 10 ; 11 ; 12 ];;
[ 1 ; 2 ; 10 ; 11 ; 12 ] : int list
```

It is easiest to explain how the append function works by example:

```
append[ 1 ; 2 ][ 10 ; 11 ]       →  1 . (append[ 2 ][ 10 ; 11 ])
                                      →  1 . (2 . (append[ ][ 10 ; 11 ]))
                                      →  1 . (2 . [ 10 ; 11 ])
                                      →  1 . [ 2 ; 10 ; 11 ]
                                      →  [ 1 ; 2 ; 10 ; 11 ]
```

The successive calls on append reduce the first argument down to the empty list but keep the current head in hand. As the recursion unwinds on the way back, we cons the values in hand onto the second argument one by one, but in reverse order.

Here is another presentation of append in which we have used the technique of pattern matching. We are guaranteed in the else-branch that the argument A is not empty and therefore has a head and a tail. \( h . T = A \) matches \( h \) with the head of \( A \) and \( T \) with the tail of \( A \).

```
#let rec append A B = null A => B
                 | let h . T = A in h . (append T B);
append = - : (* list -> * list -> * list)
#append [ 1 ; 2 ] [ 10 ; 11 ; 12 ];;
[ 1 ; 2 ; 10 ; 11 ; 12 ] : int list
```

**Quitting a session.** To close down a session we make a call on quit.

```
quit;;
- : (void -> void)
quit();;
Bye.
```

quit is a function that takes an argument of type void (written () in ML). Notice that the request for just the name of the function — here, quit — is taken by ML as a request to echo back its value and type. ML responds
- : (void -> void). This echoing is a very useful ML feature; if you can’t quite remember the definition of a system function or one of your own, type in its name and ML will remind you of its type. An actual call on `quit` requires the argument () as shown.

1.3 Typing and polymorphism

It is interesting to define an identity operator, \( I \), which returns its argument unchanged. The function is quite general and may be applied to arguments of any type. But what is the type of \( I \)? To find out, we start a new session which we maintain to the end of this chapter.

```
# let I x = x;;
I = - : (\* -> \*)
# I 3;;
3 : int
# I (3 > 2, 'heck of a language');;
(\true, 'heck of a language') : (bool # string)
```

When applied to an integer (pair, list), the function \( I \) returns the evaluation of the integer (pair, list). When applied to a function, it returns the function (whose “value” is printed as -). The call \( I \) `suc` returns `suc` and the call \( I \) \( I \) returns \( I \).

```
# I suc where suc n = n + 1;;
- : (int -> int)
# I I;;
- : (\* -> \*)
```

In general, if \* stands for any type, then \( I \) will return an object of type \* when supplied with an argument of type \*. This suggests that \( I \) has the type \* -> *, where ‘\*’ is a variable ranging over types. We are lead to an extension of the type system which allows us to introduce general-purpose type-variables representing type schemas like \* -> * for all types \*. We
introduce polymorphic type variables $*, **, ***, \ldots \star_1, \star_2, \ldots$ which can stand for any type in a type definition, but in a consistent way. For example, $* \rightarrow *$ is the type of a function like $\text{I}$ which can take an argument of any type, but will always return a value of the same type. Polymorphism is not available in most typed languages and in them we would have to write a distinct identity function for each argument type.

Here are examples of some of the types we have already introduced:

<table>
<thead>
<tr>
<th>Built-in</th>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>$(\text{bool} \ # \ * \ # \ *) \rightarrow *$</td>
<td>$(3 &gt; 2 \Rightarrow 'a' \</td>
</tr>
<tr>
<td>$\text{fst}$</td>
<td>$(* \ # \ **) \rightarrow *$</td>
<td>$\text{fst}(3, \text{true}) = 3$</td>
</tr>
<tr>
<td>$\text{snd}$</td>
<td>$(* \ # \ **) \rightarrow **$</td>
<td>$\text{snd}(3, (1, 0)) = (1, 0)$</td>
</tr>
<tr>
<td>$\text{hd}$</td>
<td>$(* \ list \rightarrow *$</td>
<td>$\text{hd} ['&lt;'; '='; '&gt;'; '] = '&lt;'$</td>
</tr>
<tr>
<td>$\text{tl}$</td>
<td>$(* \ list \rightarrow * \ list$</td>
<td>$\text{tl} ['&lt;'; '='; '&gt;'; '] = ['=; '='; '&gt;'; ']$</td>
</tr>
<tr>
<td>$\text{null}$</td>
<td>$(* \ list \ # \ * \ list \rightarrow * \ list$</td>
<td>$\text{null} [] = \text{true}$</td>
</tr>
<tr>
<td>$=$</td>
<td>$* \rightarrow *$</td>
<td>$([1; 2; 3] = []) = \text{false}$</td>
</tr>
</tbody>
</table>

IF expressions take three arguments, the first of which is a boolean. The second and third arguments must be of the same type (e.g., both boolean, both integer, both integer lists, etc). That type is the type of the result.

The most general type of a pair is $(* \ # \ **)$. We use type variables $*$ and $**$ to freely and independently range over all possible types. Thus the components of a pair may be of different types, but they may also be of the same type. The argument to $\text{fst}$ or $\text{snd}$ must always be a pair. The type of $\text{fst} \ p$ is always the type of the first component of $p$, and the type of $\text{snd} \ p$ is always the type of the second component of $p$.

```ml
# let swapPAIR (p, q) = (q, p);
swapPAIR = - : ((* # **) -> (** # *))

# swapPAIR (5, true);
(true, 5): (bool # int)

# swapPAIR (1, hd);
((-, ->) : ((* list -> *) # (** -> **))
```

The types of the built-in list operations should now be straightforward. Note that ML will not allow you to construct a list of mixed types. The only way to build a list is through the cons operation, ($@$ is defined in terms of cons), and cons glues an item of type $*$ onto a list of type $* \ list$, thus guaranteeing that all elements of a list have the same type.
Higher order functions

map is a higher order function which takes a function and a list as arguments, and goes down the list item by item applying the function to each item in turn and returning a list of the results. Clearly, the first argument to map is a function, and both the second argument and the result are lists. Let the second argument have type \( \ast \) list and the result be of type \( \ast^\ast \) list. Then the first argument is a function which is constrained to take an argument of type \( \ast \) and deliver a result of type \( \ast^\ast \).
The example shows **map** being used to double the value of each item in a list. Below, we define a predicate **even**. The first attempt at a definition fails, since we forgot to test for the possibility of a negative argument. The error message is typical for a non-terminating recursion; this HOL system was built upon both ML and Lisp and one gets the standard Lisp message (and trace) when the memory stack has overflowed.

```
#letrec even n = (n = 0) => true | not even(n-1);
   even = - : (int -> bool)

#even 0;;
true : bool

#even (-5);;
```

We correct this defect and then map **even** over a list.

```
#letrec even n = (n < 0) => even(-n)
   | (n = 0) => true
   | not even (n+1);;
   even = - : (int -> bool)

#map even [ -1; 0; 1; 2; 3; 4 ];;
[false; true; false; true; false; true] : bool list
```

The next two examples double the numbers in a list and then test each result for evenness. Not surprisingly, the result is a list each element of which is true. The second of these examples uses the built-in infix composition operator **o**, defined by \((f \circ g) x = f(g x)\), e.g. \((\text{even} \circ \text{dble}) x = \text{even} (\text{dble} x)\). Instead of mapping down the list twice, we may now map down the list just once, applying **even o dble** to each element of the list in turn.

```
#map even (map dble [ 1; 2; 3; 4 ]);;
[true; true; true; true] : bool list

#map (even o dble) [ 1; 2; 3; 4 ];;
[true; true; true; true] : bool list
```
Our second higher order example is \texttt{filter}. \texttt{filter p L} applies a predicate \( p \) to each member \( x \) of the list \( L \) and returns a list consisting of only those members of \( L \) for which \( p x \) is true.

\begin{verbatim}
#letrec filter p L = null L => []
  | p (hd L) => (hd L) . (filter p (tl L))
  | (filter p (tl L));;
filter = - : ((* -> bool) -> * list -> * list)
#let L2 = [-3; -2; -1; 0; 1; 2; 3; 4 ];;
L2 = [-3; -2; -1; 0; 1; 2; 3; 4] : int list
#filter even L2;;
[-2; 0; 2; 4] : int list

We close this section by giving an example showing that, in this case at least, if you filter twice the order of the predicates is immaterial. (The proof of the general case is set as an exercise.)

\begin{verbatim}
#let pos n = n > 0;;
pos = - : (int -> bool)
#filter even (filter pos L2);;
[2; 4] : int list
#filter pos (filter even L2);;
[2; 4] : int list
\end{verbatim}

We can filter twice with only one pass if we write a trivial function:

\begin{verbatim}
#let ep x = even x & pos x;;
ep = - : (int -> bool)
#filter ep L2;;
[2; 4] : int list
\end{verbatim}

which leads us nicely into the \( \lambda \) expressions which enable us to create use-once functions on-the-fly.

\section{\( \lambda \) expressions}

\( \lambda \) expressions\textsuperscript{3} are built into ML. The expression \( \lambda x . \epsilon \) is equivalent to a function with a formal parameter \( x \) and a body \( \epsilon \). For example

\textsuperscript{3} See also chapter 4.
\[ f \; x = \text{expr} \overset{\text{def}}{=} \lambda \; x . \; \text{expr} \]
\[ g \; x \; y = \text{expr} \overset{\text{def}}{=} \lambda \; x . \; \lambda \; y . \; \text{expr}, \; \text{or} \; \lambda \; x \; y . \; \text{expr} \]

Thus we can write \( \text{suc} = \lambda \; n . \; n + 1 \) and \( \text{add} = \lambda \; x \; y . \; x + y \). In the main, lambda expressions are evaluated by the rule

\[ (\lambda \; x . \; \text{body}) \; \text{arg} = \{ \text{arg}/x \} \; \text{body}, \]

where \( \{ \text{arg}/x \} \; \text{body} \) means substitute \text{arg} for free \( x \) throughout the \text{body} \(^4\).

As an example

\[
\text{add} \; 3 \; (a \times b) = (\lambda \; x . \; \lambda \; y . \; x + y) \; 3 \; (a \times b) \\
= (\{3/x\}(\lambda \; y . \; x + y))(a \times b) \\
= (\lambda \; y . \; 3 + y)(a \times b) \\
= [a \times b / y](3 + y) \\
= 3 + (a \times b)
\]

Here is a little ML session using \( \lambda \) expressions. Both ML and HOL use a single backslash \( \backslash \) to represent \( \lambda \).

```
#\x. x+1;;
-: (int -> int)

#it 4;;
% (\ x . \ x + 1) \ 4 -> 5 %
5 : int
```

The special ML identifier \textit{it} can be used any time to refer to the value of the last evaluation carried out by ML, except that it is not affected by \textit{let}'s and \textit{letrec}'s.

```
#(\ x. (\ y . x*x + y*y));;
-: (int -> int -> int)

#it 3;;
% a partial evaluation -> \ y . 3*x + y*y %
-: (int -> int)

#it (4+5);;
% (\ y. 3*x + y*y) \ (4+5) -> 90 %
90 : int
```

The next example gives definitions using \( \lambda \) of \textit{IF}, \textit{T} and \textit{F} and shows how they work together to mimic an \textit{IF}-expression. Note that this definition of \textit{IF} does not require that the types of the then branch and of the else branch be the same. \textit{T} and \textit{F} each take two arguments: \textit{T} throws away the second argument, \textit{F} throws away the first argument.

\(^4\)A full and proper definition of substitution is given in chapter ??.
Finally, here are examples showing how \( \lambda \) can be used to define functions on the fly. This saves the bother of defining a named function when it will only be called once. Instead, we use the \( \lambda \) notation and write it in-line.

Finally we define an in-line function to test each element of a list for both evenness and positiveness. We can then do a double filter operation with only one pass down the list.

1.6 Failure

Expressions that fail can be either reported back at the top-level or trapped in ML. \texttt{failwith} is used to signal the reason for a failure. It takes a string parameter. Here is a definition of the built-in function \( \texttt{el} \). A call \( \texttt{el \ n \ L} \) returns the \( n \)th element of the list \( L \).
Failures may be trapped by the ML construct `?`. This enables us to steer the computation in another direction. `E1 ? E2` has the value of `E1` if `E1` completes (does not fail), otherwise it takes the value of `E2` (`E1` and `E2` must either have compatible types or be `failwiths`). Here is another version of `append` to illustrate the point. The body of `append` has the form `E1 ? E2` where `E1 = ' (hd A) . (append (tl A) B)` and `E2 = B`. A call on `E1` fails if we try to take either the head or the tail of an empty list, in which case `B` is returned. This is the same gross logic as we applied the first time we wrote `append`, but this time we assume we have a non-empty first argument (and this will happen most of the time) and proceed to take it apart. When failure occurs, we recover gracefully.

```
#letrec append A B = ((hd A) . (append (tl A) B)) ? B;;
append = - : ( * list -> * list -> * list)
```

`?` will be used to great effect in the examples on datatypes in the remainder of this chapter and in the next chapter.

### 1.7 ML datatypes

The ability to define new datatypes and functions over them is a hallmark of a good programming language. ML permits the definition of both concrete and abstract datatypes. With recursive type definitions, the datatype and the functions over it are defined separately. In abstract datatypes (see next section), the type definition and the functions over it are packaged together.

ML datatypes are equality types and may be compared with `=`.

We start by defining a datatype `Aexp` to represent arithmetic expressions constructed from integer constants. The operations we choose to represent are: unary minus, `+`, `-`, `*`, and `/ (integer division). Here are some examples of how we will represent arithmetic expressions:

```
#letrec Aexp = int
```

---

5The example is taken from [59] where it is attributed to Philippe le Chenadoc.
### 1.7. ML Datatypes

**mathematics**

<table>
<thead>
<tr>
<th>expressions of type Aexp</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 NUM 5</td>
</tr>
<tr>
<td>-12 NEG(NUM 12)</td>
</tr>
<tr>
<td>3 + 5 PLUS(NUM 3, NUM 5)</td>
</tr>
<tr>
<td>2 + 3 * 5 PLUS(NUM 2, TIMES(NUM 3, NUM 5))</td>
</tr>
<tr>
<td>(2 + 3) * 5 TIMES(PLUS(NUM 2, NUM 3), NUM 5)</td>
</tr>
</tbody>
</table>

An Aexp may be one of a number of disjoint cases: a constant, a unary negation, an addition, ... etc. Each case is given a unique constructor (a tag) so that we may distinguish amongst them. We define the (recursive) datatype by listing each constructor together with the types of its fields.

```ml
#rectype Aexp
  = NUM of int
  | NEG of Aexp
  | PLUS of Aexp # Aexp
  | MINUS of Aexp # Aexp
  | TIMES of Aexp # Aexp
  | QUOT of Aexp # Aexp;;

New constructors declared:
  NUM : (int -> Aexp)
  NEG : (Aexp -> Aexp)
  PLUS : ((Aexp # Aexp) -> Aexp)
  MINUS : ((Aexp # Aexp) -> Aexp)
  TIMES : ((Aexp # Aexp) -> Aexp)
  QUOT : ((Aexp # Aexp) -> Aexp)
```

```ml
#let x = NUM 5 and y = NEG(NUM 7);;
x = NUM 5 : Aexp
y = NEG(NUM 7) : Aexp

#let a1 = PLUS(x, y) and a2 = QUOT(y, x);;
a1 = PLUS((NUM 5), NEG(NUM 7)) : Aexp
a2 = QUOT((NEG(NUM 7)), NUM 5) : Aexp
```

We now define some functions over the datatype: namely showA, which is used to “pretty print” a term of type Aexp; and evalA, which is used to evaluate an Aexp. Note that we may use = to see if two Aexp's have identical structures.

The function showA returns a string representing an Aexp in infix form. string_of_int is a built-in ML function which maps an integer argument to its decimal string representation. Strings are concatenated together with the infix ‘hat’ operator ‘^’. There are two auxiliary functions: bkt puts parentheses around its string argument, and showBin turns a binary Aexp
into a string. `showA` is defined by cases over the each of the constructors in turn. Instead of writing

```plaintext
letrec f e =
  e = e1 => res1
| e = e2 => res2
...
| e = en-1 => resn-1
| resn
```

we may instead write the rather more crisp

```plaintext
letrec f e =
case e of
  e1 . res1
| e2 . res2
...
| en . resn
```

The cases are separated by a vertical bar `|`. On the left of each case is a pattern involving a constructor and “typical” arguments. This is followed by a separating dot and then the value of the function should the argument match the case on the left.

```plaintext
#string_of_int;;
- : (int -> string)

#(string_of_int 15, string_of_int (-12));;
('15', '-12') : (string * string)
```

```plaintext
#let bkt s = ( '(' '-' s '-' ')');;;
bkt = - : (string -> string)

#letrec showBin M op N = bkt ((showA M) + op + (showA N))

and showA exp
  = case exp of
    NUM(n) . string_of_int n
  | EXP(M) . bkt ('-' '-' + (showA M))
  | PLUS(M, N) . showBin M + op + N
  | MINUS(M, N) . showBin M - op - N
  | TIMES(M, N) . showBin M '*' op * N
  | QUOT(M, N) . showBin M '/' op / N;
showBin = - : (Aexp -> string -> Aexp -> string)
showA = - : (Aexp -> string)
```

```plaintext
#map showA [ x; y; a1; a2 ];;
['5'; '--7'; ' +(5-7)'; '/ (--7)/5'] : string list
```
Aside: With \texttt{letrec} already being used for the list cons operation, defining functions over lists by cases can get rather confusing, witness

\begin{verbatim}
letrec append A B = ( case A of
    [] . B
  | (h,T) . (h . (append T B))
)
\end{verbatim}

which is why we waited until now to introduce cases.

\texttt{evalA} traverses an expression and evaluates it (the leaves are all constants). As with \texttt{showA}, \texttt{evalA} is defined by cases over the constructors. Note how we use \texttt{?} to trap division by zero and emit a pertinent error message.

\begin{verbatim}
#letrec evalA exp
  = ( case exp of
    NUM(n) . n
  | NEG(N) . -(evalA N)
  | PLUS(M,N) . (evalA M) + (evalA N)
  | MINUS(M,N) . (evalA M) - (evalA N)
  | TIMES(M,N) . (evalA M) * (evalA N)
  | QUOT(M,N) . ( (evalA M) / (evalA N) )
    ? failwith 'EVALA --- division by zero'
  );;

let exp = (QUOT(x, NUM 0));;

evalA (QUOT(x, NUM 0));;
evaluation failed EVALA --- division by zero

#(a1 = a1, a1 = a2, a2 = a2);
(true, false, true) : (bool # bool # bool)

#let L = PLUS(NUM 5, NUM 3) and R = PLUS(NUM 3, NUM 5)

(L = R, evalA L = evalA R);
(false, true) : (bool # bool)
\end{verbatim}

Having seen how repetitious are the definitions of functions over this datatype, you may be wondering if there are more compact ways of defining a datatype for arithmetic expressions. Before going on to the next section, you are encouraged to complete exercise 7 which will cause you to investigate a more condensed style of datatype definition, and lead you into the examples in the next section and in the next chapter.
1.8 Abstract datatypes

Instead of defining a datatype and its attendant operations separately, ML allows us to define an abstract datatype which encapsulates both. An abstract (recursive) datatype declaration has the form

\[
\text{abstracttype } \text{NAME} = \text{TYPE with BODY}
\]

where \( \text{NAME} \) is the name of the abstract type being defined, \( \text{TYPE} \) gives the sum type representation of \( \text{NAME} \), and \( \text{BODY} \) is a binding specifying the implementation of user defined operations over the new type in terms of the representation. The idea behind an abstract datatype is that the scope of its constructors is restricted to its body. Outside the body, they can be accessed only through given operations located inside the body and hence within the scope of the constructors. This shielding property turns out to be very important in the implementation of theorem provers, where it can be used to ensure the soundness of chains of reasoning.

We present the idea in stages via three examples of increasing scope.

1.8.1 Complex numbers

We define a datatype for complex numbers with \text{int} arguments (our version of ML does not support the type \text{real}). The recursive definition is straightforward

\[
\text{abstracttype } \text{COMPLEX} = \text{COMP} \text{ of int # int}
\]

in which we represent the complex number \( x + iy \) by \text{COMP}(x, y). Here is the same notion expressed as an abstract datatype.

```plaintext
#abstracttype COMPLEX
 = (int # int)
with
   mk_COMPLEX (x, y) = abs_COMPLEX(x, y)
   and dest_COMPLEX p = rep_COMPLEX p;;
mk_COMPLEX = - : ((int # int) -> COMPLEX)
dest_COMPLEX = - : (COMPLEX -> (int # int))

#let p1 = mk_COMPLEX(3, 4) and p2 = mk_COMPLEX(5, 6);
p1 = - : COMPLEX
p2 = - : COMPLEX

#(p1 = p1, p1 = p2, p2 = p2);
(true, false, true) : (bool # bool # bool)
```
abs\_COMPLEX and \texttt{rep\_COMPLEX} are special functions, supplied by ML, with which to access the sum type. In general, these names are manufactured by ML by concatenating the name of the abstract datatype (here \texttt{COMPLEX}) onto abs\_ and \texttt{rep\_}. These special functions are used to translate between the user notation and the ML sum type representation:

- \texttt{abs\_COMPLEX(a, b)} maps the user supplied integer pair (a, b) to the internal representation of a complex number.

- \texttt{rep\_COMPLEX p} maps the internal representation of a complex number \( p \) to its co-ordinates as a pair of integers.

Clearly \texttt{rep\_COMPLEX (abs\_COMPLEX(a, b))} \neq (a, b).

\begin{verbatim}
#rep\_COMPLEX;;

unbound or non-assignable variable rep\_COMPLEX
1 error in typing
typecheck failed
\end{verbatim}

\texttt{rep\_COMPLEX} and likewise \texttt{abs\_COMPLEX} are unbound outside the body of the abstract datatype. We must supply our own access routines to build and take apart complex number objects. Clearly they must build upon \texttt{abs\_COMPLEX} and \texttt{rep\_COMPLEX} and therefore must be textually positioned within the body of the type declaration for \texttt{COMPLEX}. By convention, we will prefix these function definitions by \texttt{mk\_} and \texttt{dest\_} respectively. We define \texttt{mk\_COMPLEX} to construct a complex number from two integer arguments. We also define \texttt{dest\_COMPLEX} to access a \texttt{COMPLEX} object and return its arguments. Both functions are trivial rewrites of the basic supplied operations, but have a wider scope.

We can elaborate the notion of a complex number and write a richer set of functions to manipulate them, but we can only do so using our \texttt{mk\_} and \texttt{dest\_} operations. As examples, here are definitions of functions to "add" two complex numbers together and to show a complex number.
### Program 1.1.5

The following program defines operations on complex numbers:

```ml
#let addCOMPLEX p1 p2 = let (a1, b1) = dest_COMPLEX p1
  and (a2, b2) = dest_COMPLEX p2
  in
    mk_COMPLEX (a1 + a2, b1 + b2);
addCOMPLEX = - : (COMPLEX -> COMPLEX -> COMPLEX)

#let p3 = addCOMPLEX p1 p2;;
p3 = - : COMPLEX

#let showC C = let (p, q) = dest_COMPLEX C in
  ("('" + (string_of_int p) + " + i" + (string_of_int q) + ")");;
showC = - : (COMPLEX -> string)

#showC p3;;
"(8 + i10)"
```

In the development of an abstract datatype, we usually start with a bare bones definition and embellish it with extra operations textually sited outside its body. Once the full definition has been worked out, you may wish to recompile it as a rounded entity with all its functionality embedded in its body, as below:

```ml
#abstracttype COMPLEX = (int # int)
with
    mk_COMPLEX (x, y) = abs_COMPLEX(x, y)
  and dest_COMPLEX p = rep_COMPLEX p
and addCOMPLEX p1 p2 = let (a1, b1) = rep_COMPLEX p1
  and (a2, b2) = rep_COMPLEX p2
  in abs_COMPLEX(a1+a2, b1+b2)
and subCOMPLEX p1 p2 = let (a1, b1) = rep_COMPLEX p1
  and (a2, b2) = rep_COMPLEX p2
  in abs_COMPLEX(a1-a2, b1-b2)
and multCOMPLEX p1 p2 = let (a1, b1) = rep_COMPLEX p1
  and (a2, b2) = rep_COMPLEX p2
  in abs_COMPLEX(a1*a2-b1*b2, a1*b2+a2*b1)
and showC C = let (p, q) = rep_COMPLEX C
  in ("('" + (string_of_int p) + " + i" + (string_of_int q) + ")");;
mk_COMPLEX = - : ((int # int) -> COMPLEX)
dest_COMPLEX = - : (COMPLEX -> (int # int))
addCOMPLEX = - : (COMPLEX -> COMPLEX -> COMPLEX)
subCOMPLEX = - : (COMPLEX -> COMPLEX -> COMPLEX)
multCOMPLEX = - : (COMPLEX -> COMPLEX -> COMPLEX)
showC = - : (COMPLEX -> string)
```
1.8. ABSTRACT DATATYPES

1.8.2 Integers: I

When we define an abstract datatype, the sum type will generally have several component types, and the translation process into and out of the sum type is complicated by our having to select the correct constructor. We put the idea across by considering primitive integer expressions (either constants or variables) whose datatype definition is

\[ \text{rectype PRIM} = \text{CON of int} \mid \text{VAR of string} \]

Typical primitive expressions are CON 5 and VAR 'x'.

As an abstract datatype, this takes the form

\[ \text{absrectype PRIM} = \text{int} + \text{string} \]

Inside the body of the abstract datatype, we have to define both a function which makes an object of that type and a function which takes an object of that type apart, for each of the separate types in the sum type.

Think of the internal representation of the sum type as a pair of types \((\text{left}, \text{right})\) where constants are the left type and variables are the right type. When injecting a value into a \(\text{PRIM}\) datatype, we first select which field — the left or the right — using the appropriate built-in function, \text{inl} or \text{inr} respectively (read \text{inl} as “in left” and \text{inr} as “in right”) and then translate to the internal representation by passing that to \(\text{abs\_PRIM}\). That is we inject a constant value \(n\) into the internal representation by a call \(\text{abs\_PRIM}(\text{inl } n)\), and inject a variable \(v\) into the internal representation by a call \(\text{abs\_PRIM}(\text{inr } v)\). Type checking ensures that the data being injected has a type consistent with the definition of \(\text{PRIM}\), that is an integer on the left and a string on the right.

Similarly, we can extract data from the internal representation by first calling \(\text{rep\_PRIM}\) and then selecting that value as a left type \(\text{outl}\) or a right type \(\text{outr}\). Here is a definition of \(\text{PRIM}\) with some simple examples showing its shortcomings.

```hs
#absrectype PRIM
  = int + string
with
  mk_CON n = abs\_PRIM(inl n)
  and dest_CON P = outl(rep\_PRIM P)
  and mk_VAR v = abs\_PRIM(inr v)
  and dest_VAR P = outr(rep\_PRIM P);
  mk_CON = - : (int \rightarrow \text{PRIM})
  dest_CON = - : (\text{PRIM} \rightarrow \text{int})
  mk_VAR = - : (string \rightarrow \text{PRIM})
  dest_VAR = - : (\text{PRIM} \rightarrow \text{string})
```
First we should ensure that only single-letter variables are allowed. Second the attempt to take apart a constant in a call on dest_VAR is caught by the system, but the error message is less than enlightening. We use failure to trap unwanted arguments to dest_CON and dest_VAR and emit useful error messages.

Enforcing single-letter arguments to mk_VAR requires a little work on the side. Unfortunately, ML does not have either of the operators ≤ or ≥ built-in. Neither does it allow us to compare strings with < or >. We first collect together some some auxiliary definitions:

- **leq** and **geq** enable us to compare integers.

- **between** checks to see if a character lies in a certain range, e.g., between 'a' x 'z' returns true iff x is a lower case letter. The function ascii_code: string → int is built into ML. It is an error if the string is empty; if not it returns the ascii code of the first character in the string.

- **isLetter** returns true if its single character argument is either a lower case letter or an upper case letter; false otherwise.
### 1.8. Abstract Datatypes

```ocaml
#let leq a b = a < b or a = b
and geq a b = a > b or a = b;;
leq = -: (int -> int -> bool)
geq = -: (int -> int -> bool)

#ascii_code;;
- : (string -> int)

#ascii_code 'a';;
97: int

#ascii_code 'ab';;
97: int

#let between a x b
  = (leq (ascii_code a) (ascii_code x))
    & (leq (ascii_code x) (ascii_code b));;
between = -: (string -> string -> string -> bool)

#let isSingleChar x = length(explode x) = 1;;
isSingleChar = -: (string -> bool)

#let isLetter x
  = isSingleChar x & (between 'a' x 'z') or (between 'A' x 'Z');;
isLetter = -: (string -> bool)

#map isLetter ['a'; '7'; 'ab'];;
[true; false; false] : bool list
```

Now we can code a better version of PRIM.

```ocaml
#absrectype PRIM
  = int + string
with
  mk_CON n = abs_PRIM(inl n)
and dest_CON P = outl(rep_PRIM P)
  ? failwith 'dest_CON expects a CON'

  and mk_VAR v = isLetter v => abs_PRIM(inr v)
  | failwith 'mk_VAR takes a single letter'
and dest_VAR P = outr(rep_PRIM P)
  ? failwith 'dest_VAR expects a VAR';;

mk_CON = - : (int -> PRIM)
dest_CON = - : (PRIM -> int)

mk_VAR = - : (string -> PRIM)
dest_VAR = - : (PRIM -> string)

#let v = mk_VAR 'v' and n = mk_CON 5;;
v = - : PRIM
n = - : PRIM
```
#(dest_VAR v, dest_CON n);;
('v', 5) : (string # int)

#mk_VAR '5 = 7';;
evaluation failed     mk_VAR takes a single letter

#mk_VAR '9';;
evaluation failed     mk_VAR takes a single letter

#dest_VAR n;;
evaluation failed     dest_VAR expects a VAR

Notice our use of failwith to pin point where things have gone wrong. Another elegant use of failure is to select amongst the constructors of a sum type. Here is a function to show a PRIM:

#let showP P =
  (let n = dest_CON P in (string_of_int n))
? (dest_VAR v);
showP = - : (PRIM -> string)

#map showP [ n;; v ];;
['5'; 'v'] : string list

The let expression is always evaluated. If the argument to showP is a constant, it is taken apart and its integer argument is shown (as a string). If the argument to showP is a variable, dest_CON will fail and control is steered to the second expression dest_VAR v. The argument is then destroyed and its string argument returned.

1.8.3 Integers: II

We complete our presentation of abstract data types by writing a datatype for constant integer expressions (no variables this time). Expressed recursively, the datatype reads

rectype NSPRIM = CON of int
| OP1 of string # NSPRIM
| OP2 of string # NSPRIM # NSPRIM

Here is the corresponding type in our abstract representation for NSPRIM

int + (string # NSPRIM) + (string # NSPRIM # NSPRIM)
which we think of as a nested pair \((L, (RL, RR))\). The constant type is on the left \(L\), and the unary and binary types are on the right. The unary type in the right pair on the left \(LR\), and the binary type is in the right pair on the right \(RR\). However complicated the sum type being defined, it is clear that we can get at the constructors we want with the right combinations of the \(\text{inl}/\text{inr}\) and \(\text{outl}/\text{outr}\) primitives. The notation is systematic but messy — one just has to be careful. This is where the pattern \((L, (RL, RR))\) comes in handy as a mnemonic aid. It indicates the pattern of in calls required to get a specific constructor. The individual patterns are “mirrored” for the \(\text{out}\) calls (here they come out as \((L, (LR, RR))\)).

```plaintext
#absrectype NSPRIM
  = int + (string # NSPRIM) + (string # NSPRIM # NSPRIM)
with
  mk_CON n = abs_NSPRIM(inl n)
  and dest_CON t = outl(rep_NSPRIM t)
  ? failwith 'dest_CON expects a CON'
  and mk_OP1 (op, t) = abs_NSPRIM(inr(inl(op, t)))
  and dest_OP1 t = outl(outr(rep_NSPRIM t))
  ? failwith 'dest_OP1 expects an OP1'
  and mk_OP2 (op, t1, t2) = abs_NSPRIM(inr(inr(op, t1, t2)))
  and dest_OP2 t = outr(outr(rep_NSPRIM t))
  ? failwith 'dest_OP2 expects an OP2'

mk_CON = - : (int -> NSPRIM)
dest_CON = - : (NSPRIM -> int)
mk_OP1 = - : ((string # NSPRIM) -> NSPRIM)
dest_OP1 = - : (NSPRIM -> (string # NSPRIM))
mk_OP2 = - : ((string # NSPRIM # NSPRIM) -> NSPRIM)
dest_OP2 = - : (NSPRIM -> (string # NSPRIM # NSPRIM))
```

The body of abstract datatypes can be made to look a little neater if we use the composition operator \(\circ\) and partial applications. Compare and contrast the above with
The details are of little significance (the \(o\) representation will be slower). Choose whichever style suits you.

1.8.4 Structural induction

Now that we can know how to define datatypes and functions over them, we need a technique for their verification\(^6\). To explain the idea, we take the (typical) datatype

\[
\text{rectype } D = C_0 \mid C_1 \text{ of } D \mid C_2 \text{ of } D \# D
\]

and consider the set of all expressions we may form from these constructors classified by their \textit{depth}. Depth is defined recursively:

1. the simplest expression, of depth 0, is formed from the constructor \(C_0\).

2. expressions of depth \(n + 1\) are formed from expressions of depth \(n\) or less by a single application of a constructor.

As examples, the expression of depth 0 is \(C_0\), the expressions of depth 1 are \(C_1(C_0), C_2(C_0, C_0)\) and those of depth 2 are

\[^6\text{This account is based upon [94, pages 33–37, 111–113, 182–184] which is recommended for a clear introduction to structural induction.}\]
Each new value is constructed either from no values of the type (depth 0) or from strictly simpler subexpressions of the same type which carry all the information about their own construction. These facts, together with the fact that all expressions are finite enables us to state a general principle of structural induction based upon the depth (or complexity) of subexpressions. For the datatype D with constructors C₀, C₁, and C₂, structural induction allows us to prove that a property P holds for all values of type D by establishing that:

1. \( P \ C₀ \) holds
2. \( P \ (C₁ \ d) \) holds assuming \( P \ d \) holds
3. \( P \ (C₂ (d₁, d₂)) \) holds assuming that both \( P \ d₁ \) and \( P \ d₂ \) hold.

In general, datatype definitions may include an arbitrary number of constructors with no arguments (like \( C₀ \)), one argument (like \( C₁ \)), two arguments (like \( C₂ \)), three arguments etc. Our stated principle generalises in the obvious manner.

Here are datatype definitions for natural numbers, lists, and trees, each with a simple function defined over it. Our datatype for trees has two constructors of depth 0.

```
rectype nat = ZERO | SUC of nat

add ZERO n = n  \hspace{1cm} (i)
add (SUC m) n = SUC (add m n) \hspace{1cm} (ii)
```

```
rectype * list = [] | CONS of * # * list

map f [] = []  \hspace{1cm} (i)
map f (a::A) = CONS (f a) (map f A) \hspace{1cm} (ii)
```
rectype * tree = EMPTY | LEAF of * | NODE of * tree

rev EMPTY = EMPTY
rev (LEAF n) = LEAF n
rev (NODE(L, n, R)) = NODE(rev R, n, rev L)

If we are to do proofs of properties over these datatypes, the principle of structural induction tells us that,

1. For nats. Prove (i) \( P \) \( \text{ZERO} \) and (ii) \( P \) \( \text{SUC} \) \( m \) assuming \( P \) \( m \) holds.

2. For lists. Prove (i) \( P \) \([\]\) and (ii) \( P \) \( \text{CONS} \) \( a \) \( A \) for all \( a \) assuming \( P \) \( A \) holds.

3. For trees. Prove (i) \( P \) \( \text{EMPTY} \), (ii) \( P \) \( \text{LEAF} \) \( n \) for all \( n \), and (iii) \( P \) \( \text{NODE} \) \( \langle L, n, R \rangle \) for all \( n \) assuming \( P \) \( L \) and \( P \) \( R \) holds.

When defining a function over a recursive datatype, we essentially do a case analysis over the constructors. Each case in the definition will be used in a case in the proof.

Example 1.8.1 A mapping theorem

Use structural induction to prove \( \text{map} \ f \ (\text{map} \ g \ L) = (\text{map} \ (f \circ g) \ L) \).

**Proof:**

**Base case:** \( P \) \([\]\)
\[
\text{map} \ f \ (\text{map} \ g \ [\]) \ = \ \text{map} \ f \ [\] \ = \ [\] \ = \ \text{map} \ (f \circ g) \ [] \quad \text{unfold with map (i)}
\]

**Induction step:** \( P \) \( \text{CONS} \) \( a \) \( L \)
**Assume** \( \text{map} \ f \ (\text{map} \ g \ L) = (\text{map} \ (f \circ g) \ L) \)
\[
\text{map} \ f \ (\text{map} \ g \ \langle \text{CONS} \ a \ L \rangle) \ = \ \text{map} \ f \ \langle \text{CONS} \ (g \ a) \ (\text{map} \ g \ L) \rangle \quad \text{unfold with map (ii)}
\]
\[
= \ \text{CONS} \ (f (g \ a)) \ \text{map} \ f \ (\text{map} \ g \ L) \quad \text{unfold with map (ii)}
\]
\[
= \ \text{CONS} \ ((f \circ g) \ a) \ (\text{map} \ f \ (\text{map} \ g \ L)) \quad \text{Induction hypothesis}
\]
\[
= \ \text{map} \ (f \circ g) \ \langle \text{CONS} \ a \ L \rangle \quad \text{fold with map (ii)}
\]

\( \square \)
EXERCISES 1

Exercise 1.1 Write a function `between` which takes two integers as arguments and returns a list of all the numbers “between” them, e.g. `between 0 3` should return `[0; 1; 2; 3].`

Exercise 1.2 Define a function `rep:int->*->* list` which takes a non-negative counter, say `n`, and an object of type `*` and replicates the object `n`-fold in a list.

Exercise 1.3 Define a function `cp` which takes two lists and returns a list of all possible pairs constructed from the items in its argument lists (their cartesian product).

Exercise 1.4 We define a function `fold` which takes an operator, a default value, and a list of arguments, `L`. A call `fold op r []` returns the default value, `r`. A call `fold op r [ A1; A2; ... ; An ]` returns `(op A1 (op A2 ... (op An r) ... ))`.

```plaintext
#letrec fold op r L
  = null L => r
  | (op (hd L) (fold op r (tl L)))
fold = - : ((* -> ** -> **) -> ** -> * list -> **)[[fold (\ x y . (x + y)) 0 [1; 2; 3; 4]]];
10 : int
#let some = fold (\ x y . (x or y)) false;
some = - : (bool list -> bool)
some [true; false; true]];
true : bool
```

In the manner of `some` define a function `add` to add elements of an integer list together, a function `mult` to multiply elements of an integer list together, and a function `all` which operates on a boolean list and returns true only if all the latter’s elements are true.

Exercise 1.5 “Pebbles” may be red, white, or blue. Write a function to sort a list of pebbles in arbitrary order into a list with the red ones first, then the white ones, then the blue ones.

Exercise 1.6 Given a pile of `n` coins, with faces called heads and tails. Initially all the coins in the pile are heads up. The `flip` operation takes the top `m` coins (1 <= m <= n) turns it over and puts it back. The `cycle` operation carries out `flip 1, flip 2, ..., flip n`. After each cycle, we check to see if the pile is all heads again, and repeat the cycle operation until it is. Program the flip, cycle, and check operations.
Exercise 1.7 Write functions \texttt{show} and \texttt{eval} functions over the datatype
\begin{verbatim}
 rectype Aexp2 = NAT of int |
                UNARY of string # Aexp2 |
                BINARY of string # Aexp2 # Aexp2
\end{verbatim}

In this representation, we would represent $-5$ by \texttt{UNARY ' ' (NAT 5)} and $4 + 5$ by \texttt{BINARY ' + ' (NAT 4) (NAT 5)}.

Note that this compacted representation has only three constructors as against the six required in \texttt{Aexp} and most functions over the compact representation have only 3 cases instead of 6.

Discuss the pros and cons of these styles of representation.

Exercise 1.8 Write a datatype for manipulating boolean expressions comprised of the constants true and false, and the logical operators not, and, or, implies and equivalent. Write functions \texttt{show} and \texttt{eval} functions over this datatype.

Exercise 1.9 Write functions \texttt{show} and \texttt{eval} functions over the abstract data type \texttt{NSPRIM}.

Exercise 1.10 Can you prove the following properties of \texttt{filter}?

1. $\text{filter } p \ (\text{filter } q \ L) = \text{filter } q \ (\text{filter } p \ L)$
2. $\text{filter } p \ (\text{filter } q \ L) = \text{filter } (\lambda x . \ p x \ & \ q x) \ L$
3. $\text{filter } p \ (\text{map } f \ L) \neq \text{filter } (p \ o \ f) \ L$

Is there a way to achieve the effect of \texttt{map } f \ (\text{filter } p \ L) with only one pass through the list \texttt{L}? Prove it.

Exercise 1.11 Write functions to (i) insert a fresh item in a tree, (ii) print the values in a tree, (iii) count the number of leaves in a tree and (iv) to collect the leaves of a tree in a list.

Exercise 1.12 Use structural induction to prove that \texttt{rev(rev T)} = \texttt{T} for all trees \texttt{T}. 
Chapter 2
Terms in \( p \text{HOL} \)

The next two chapters take their inspiration and form from a well written and insightful paper by Mike Gordon [40]. We want to put across a number of important techniques: the axiomatization of a logic, conducting proofs in a logic, and how to implement a proof assistant in ML. Instead of working with full-blown higher order logic, we illustrate the ideas with \( p \text{HOL} \), a propositional subset of HOL. The techniques we show extrapolate to HOL.

In this chapter we remind you of propositional logic and show you an axiomatisation. We use ML to define an abstract datatype for terms (well formed boolean expressions). The abstract datatype encompasses constants, variables, negation, and the four binary boolean operators \( \land \), \( \lor \), \( \supset \) and \( \equiv \). We also write a simple parser for these terms.

2.1 \( p \text{HOL} \) — the propositional subset of HOL

Propositional logic is a notation in which we can write terms that evaluate to either true or false. Terms are boolean expressions that are constructed systematically from primitives—constants, variables, operators—and parentheses.

- the logical constants are \( T \) standing for true and \( F \) standing for false
- logical variables are constructed from letters, digits, primes and underscores but start with a letter. E.g. \( x \), \( xor2spec \), \( nAdder\_imp \). A logical variable has either the value \( T \) or the value \( F \).
- the five logical operators are \( \neg \) (not), \( \land \) (and), \( \lor \) (or), \( \supset \) (implies, or if-then), and \( \equiv \) (equivalent, or iff) with interpretations

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>( \neg a )</th>
<th>a ( \land b )</th>
<th>a ( \lor b )</th>
<th>a ( \supset b )</th>
<th>a ( \equiv b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
</tbody>
</table>

Terms in the propositional logic are sequences of these symbols constructed according to certain simple rules of formation:
• logical constants and logical variables on their own are terms
• if $P$ is a term, then so is $\neg P$
• if $P$ and $Q$ are terms, then so are $P \land Q$, $P \lor Q$, $P \supset Q$, and $P \equiv Q$
• if $P$ is a term, then so is $(P)$.

Here are a couple of examples (with extra bracketing provided to make their meanings clearer): $(a \equiv b) \equiv ((a \land b) \lor (\neg a \land \neg b))$ may be read as “$a$ is equivalent to $b$ iff either $a$ and $b$ are both true or $a$ and $b$ are both false”, and $((a \lor b) \land (b \lor c)) \supset (a \lor c)$ may be read as ‘if $a$ implies $b$ and $b$ implies $c$, then $a$ implies $c’”. In our interpretation, both statements are true.

In order to reduce the number of parentheses in statements, each logical operator has a precedence associated with it: $\neg$ binds tightest, then $\lor$, then $\land$, then $\supset$. For example $\neg a \land \neg b \lor \neg c \lor d$ is understood as $((\neg a) \land (\neg b)) \lor ((\neg c) \lor d)$.

2.2 A datatype for $\mathcal{P}$HOL terms

The first problem we have to settle concerns the keyboard representation of terms in $\mathcal{P}$HOL, particularly the operators. The most obvious route is to represent terms as ML strings and to mimic the HOL representation as closely as we can. Our representation is given in table 2.1. Note that backslash \ is treated as a special character within ML strings, for example \L is taken as newline.

<table>
<thead>
<tr>
<th>English</th>
<th>Logic</th>
<th>$\mathcal{P}$HOL representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>not</td>
<td>$\neg$</td>
<td>'!'</td>
</tr>
<tr>
<td>and</td>
<td>$\land$</td>
<td>'/'</td>
</tr>
<tr>
<td>or</td>
<td>$\lor$</td>
<td>'/'</td>
</tr>
<tr>
<td>implies</td>
<td>$\supset$</td>
<td>'=&gt;'</td>
</tr>
<tr>
<td>equivalent</td>
<td>$\equiv$</td>
<td>'&lt;=&gt;'</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>'T'</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>'F'</td>
</tr>
</tbody>
</table>

Table 2.1 Representation of terms in $\mathcal{P}$HOL

Within ML strings, a single backslash has to be written \\, hence the “funny” representations of logical or and logical and. In addition we limit ourselves to single letter identifiers. As an example the mathematical $(x \supset y) \land (y \supset x) \equiv (x \equiv y)$ becomes

'$(x \Rightarrow y) \land (y \Rightarrow x) \Rightarrow (x \leftrightarrow y)'
in our implementation of $\mu$HOL.

We now give an abstract datatype for terms.

```
#let leq a b = a < b or a = b
and geq a b = a > b or a = b;;
leq = - : (int -> int -> bool)
geq = - : (int -> int -> bool)

#let between a x b = (leq (ascii_code a) (ascii_code x))
and (leq (ascii_code x) (ascii_code b));
between = - : (string -> string -> string -> bool);

#let isSingleChar x = length(explode x) = 1;;
isSingleChar = - : (string -> bool)

#let isLetter x = isSingleChar x & (between 'a' x 'z') or (between 'A' x 'Z');;
isLetter = - : (string -> bool)

#abstracttype TERM
= bool + string + (string # TERM) + (string # TERM # TERM)
with

mk_CON b = (abs_TERM o inl) b
and dest_CON t = (outl o rep_TERM) t
? failwith 'dest_CON expects a CON'

and mk_VAR v = isLetter v => (abs_TERM o inr o inl) v
? failwith ('TERM : ^ v ^ ' not a letter')
and dest_VAR t = (outl o outr o rep_TERM) t
? failwith 'dest_VAR expects a VAR'

and mk_OP1 op a = (abs_TERM o inr o inr o inl) (op, a)
and dest_OP1 t = (outl o outr o outr o rep_TERM) t
? failwith 'dest_OP1 expects an OP1'

and mk_OP2 op a1 a2 = (abs_TERM o inr o inr o inr) (op, a1, a2)
and dest_OP2 t = (outr o outr o outr o rep TERM) t
? failwith 'dest_OP2 expects an OP2';;

mk_CON = - : (bool -> TERM)
dest_CON = - : (TERM -> bool)
mk_VAR = - : (string -> TERM)
dest_VAR = - : (TERM -> string)
mk_OP1 = - : (string -> TERM -> TERM)
dest_OP1 = - : (TERM -> (string # TERM))
mk_OP2 = - : (string -> TERM -> TERM -> TERM)
dest_OP2 = - : (TERM -> (string # TERM # TERM))
```

The type of our abstract representation for $TERM$ is

```
bool + string + (string # TERM) + (string # TERM # TERM)
```
which we think of as a nested pair \((L, (LR, (LRR, RRR)))\). The constant
type is on the left \(L\), the variable type is at \(LR\), the unary type at \(LRR\)
and the binary type at \(RRR\). The next thing we do is tailor a repertoire of \(\text{mk}\)
and \(\text{dest}\) functions to the individual operators. The \(\text{mk}\) definitions look
neater when we use the principle of extensionality which states that if two
functions \(f\) and \(g\) return the same value for the same argument, then they
represent the same function, i.e., if \(fx = gx\) for all \(x\), then \(f = g\). In ML
and HOL, we are allowed to drop the same argument from both sides of a
function definition. Compare the two equivalent definitions of \(\text{mk}_{\text{NEG}}\).

\[
\begin{align*}
\#\text{let} & \text{mk}_{\text{NEG}} s = \text{mk}_{\text{OP}1} (\sim) s ;;
\text{mk}_{\text{NEG}} = - : \text{(TERM} \rightarrow \text{TERM}) \\
\#\text{let} & \text{mk}_{\text{NEG}} = \text{mk}_{\text{OP}1} (\sim) \\\n\text{and} & \text{mk}_{\text{CONJ}} = \text{mk}_{\text{OP}2} (\&\&\&) \\
\text{and} & \text{mk}_{\text{DISJ}} = \text{mk}_{\text{OP}2} (\|\|\|) \\
\text{and} & \text{mk}_{\text{IMP}} = \text{mk}_{\text{OP}2} (\Rightarrow) \\
\text{and} & \text{mk}_{\text{EQV}} = \text{mk}_{\text{OP}2} (\Leftrightarrow) ;;
\text{mk}_{\text{NEG}} = - : \text{(TERM} \rightarrow \text{TERM}) \\
\text{mk}_{\text{CONJ}} = - : \text{(TERM} \rightarrow \text{TERM} \rightarrow \text{TERM}) \\
\text{mk}_{\text{DISJ}} = - : \text{(TERM} \rightarrow \text{TERM} \rightarrow \text{TERM}) \\
\text{mk}_{\text{IMP}} = - : \text{(TERM} \rightarrow \text{TERM} \rightarrow \text{TERM}) \\
\text{mk}_{\text{EQV}} = - : \text{(TERM} \rightarrow \text{TERM} \rightarrow \text{TERM}) \\
\end{align*}
\]

The \(\text{dest}\) operations are a little more complicated since we have to check
that the term we destroy has the expected operator and report failure if not.
It is convenient to write two generic destroy functions, and specialise
them. Again note that using extensionality buys a little extra neatness of
definition.

\[
\begin{align*}
\#\text{let} & \text{dest}_{\text{UNARY}} \text{unop message t} \\
& = (\text{let} \ (\text{op, res}) = \text{dest}_{\text{OP}1} t \text{ in} \\
& \quad \ (\text{op = unop}) \Rightarrow \text{res} \ | \ \text{fail} \\
& \quad ) \ ? \ \text{failwith message} \\
\text{and} & \text{dest}_{\text{BINARY}} \text{binop message t} \\
& = (\text{let} \ (\text{op, t1, t2}) = \text{dest}_{\text{OP}2} t \text{ in} \\
& \quad \ (\text{op = binop}) \Rightarrow \text{t1, t2} \ | \ \text{fail} \\
& \quad ) \ ? \ \text{failwith message}; \\
\text{dest}_{\text{UNARY}} = - : \text{(string} \rightarrow \text{string} \rightarrow \text{TERM} \rightarrow \text{TERM}) \\
\text{dest}_{\text{BINARY}} = - : \text{(string} \rightarrow \text{string} \rightarrow \text{TERM} \rightarrow \text{TERM} \rightarrow \text{TERM}) \\
\end{align*}
\]
We also provide a pretty printer and show it in action.

```
#let bkt t = ( '(' t ' ')' );

bkt = - : (string -> string)

#letrec showTerm t
  = ( let b = dest_CONJ t in (b => 'T' | 'F'))
  ? (dest_VAR t)
  ? (let (op, s) = dest_OP1 t in (op ` (showTerm s)))
  ? (let (op, t1, t2) = dest_OP2 t in
    bkt ( (showTerm t1) ` ` op ` ` (showTerm t2)));

showTerm = - : (TERM -> string)
```

```
#let x = mk_VAR 'x' and y = mk_VAR 'y';;

x = - : TERM

y = - : TERM

#(mk_CONJ true = mk_CONJ true, mk_CONJ true = mk_NEG3(mk_CONJ false));;

(true, false) : (bool # bool)

#let xtheny = mk_IMP x y and ythenx = mk_IMP y x;;

xtheny = - : TERM

ythenx = - : TERM

#showTerm (mk_EQV (mk_CONJ xtheny ythenx) (mk_EQV x y));;

'( (x => y) \ (y => x)) <=> (x <=> y)'; : string
```

Another function we require is subst new v t which will substitute the term new for each occurrence of the variable v in a term t. It makes heavy use of exceptions.

```
#letrec subst new v t
  = ( (dest_VAR t = dest_VAR v) => new | t )
  ? (let (op, s) = dest_OP1 t in mk_OP1 op (subst new v s))
  ? (let (op, t1, t2) = dest_OP2 t in
    mk_OP2 op (subst new v t1) (subst new v t2))
  ? t;;

subst = - : (TERM -> TERM -> TERM -> TERM)
```
2.3 Lexical analysis of \( p \text{HOL} \)

As evidenced by the example using \texttt{subst} above, it is rather tedious constructing terms by hand each time. Accordingly, we now provide a simple translator \( P \) which maps a well formed string to term datatype, e.g. we will be able to write \( P \text{ `p \Rightarrow \neg (p \land T)` } \) instead of the cluttered

\[
\text{mk IMP} (\text{mk VAR `p`) (\text{mk NEG(mk CONJ (mk VAR `p`) (mk CON true))}))
\]

The translator has two main parts:

1. lex the input string into basic tokens or atoms—these are constants, variables, operators, and parentheses
2. parse the atoms into an expression tree

The lexer is written as a single function \texttt{getATOM}. We first explode the source string into a list of characters and examine them one by one.

\[
\text{explode `x \Rightarrow x \land x` ;}
\]

\[
\text{[`x`; `\land`; `x`; `\Rightarrow`; `x` ; `\land`; `x` ; `\Rightarrow` ; `x` ; `\land` ; `x` ; `\Rightarrow` ; `x` ]
\]

: string list

Notice that \( \land \) and \( \land \), the source representations of \( \lor \) and \( \land \) respectively, are exploded into just two characters.

The task of the lexer is to process the source, discarding white space, and return the \( p \text{HOL} \) atoms it contains. It is convenient for the parser to process the \( p \text{HOL} \) atoms one by one and so we have to remember the state of the lexer in between calls on \texttt{getATOM}. We organise \texttt{getATOM} to return a pair \( \text{token, rest} \) where \texttt{token} is the next atom and \texttt{rest} is the source that remains to be lexed. For example, given the source code above, the first call on \texttt{getATOM} will return

\[
\text{`x`, [`\lor`; `\Rightarrow`; `x`; `\land`; `x` ; `\Rightarrow` ; `x` ; `\land` ; `x` ; `\Rightarrow` ; `x` ]}
\]
and the second call on getATOM will return

\[(‘=’‘, \[ ‘; ‘; ‘; ‘; ‘\])\]

getATOM is typed as \(\text{string list} \rightarrow (\text{string} \# \text{string list})\).

We first collect together some auxiliary definitions:

- \(\text{isWhite}\) checks to see if a character is white space, i.e. either a blank (which may be written as \(\text{' '}\) or \(\text{'\S'}\)), a line feed \(\text{'\L'}\), a carriage return \(\text{'\R'}\), or a tab \(\text{'\T'}\).

- It is convenient to group the possible input atoms according to their length, hence isSingle, isDouble and isTriple.

```ocaml

#let isWhite x = mem x [' ' '; '\L'; '\R'; '\T' ];
and isSingle x = mem x ['(' '; ');'];
and isDouble x = mem x ['/'\'/'; '\\/' ];
and isTriple x = mem x ['=='>'; '=/='/'; '/\'/'; '/\' ];
isWhite = - : (string -> bool);
isSingle = - : (string -> bool);
isDouble = - : (string -> bool);
isTriple = - : (string -> bool);

```

getATOM has type \(\text{string list} \rightarrow (\text{string} \# \text{string list})\) — it accepts a list of characters and returns a pair consisting of the next atom and what is left to parse. In a call to getATOM, let the head of the input list be \(x\), and the tail of the input list be \(X\). Then getATOM is structured into the following cases:

1. if \(x\) is white space, then skip it and read on by calling getATOM \(X\).
2. if \(x\) is a letter, we have a boolean constant or a variable. Return \((x, X)\).
3. look at the next three characters — \(x, y\) and \(z\). If \(x^y^z\) forms a three character operator, then return \((x^y^z, \text{tl(tl X)})\).
4. look at the next two characters — \(x\) and \(y\). If \(x^y\) forms a two character operator, then return \((x^y, \text{tl X})\).
5. if \(x\) is a single character token, then return \((x, X)\).
6. any other character is an error and the lexical analysis fails. No attempt is made to recover from lexical errors.

We assume that the list of characters to be parsed has the “shape” \(x.y.z.Z\), that is a list with \(x\) as first element, \(y\) as second element and \(z\) as third element. The \(Z\) pattern matches as the rest of the list. Since we
don’t want \texttt{getATOM} will fail if it cannot lookahead this far, we use the pair
(‘ ‘, [ ]) as a convenient default.

The cases are straightforward to code. In general, there may be some
overlap in the characters comprising basic atoms (for example, =>, =>, and
> are all legal in \textsc{HOL}). When lexing such atoms, we look for the longest
string first.

\begin{verbatim}
#let snoc X = (hd X, tl X) ? (’ ’, [ ]); snoc = _ : (string list -> (string # string list))
#letrec getATOM xX
  = null xX => (’ ’, [ ]); |
    let (x, X) = snoc xX in
    let (y, Y) = snoc X in
    let (z, Z) = snoc Y in
    isWhite x => getATOM X
    isLetter x => (x, X)
    isTriple (x’y’z’) => (x’y’z’, Z)
    isDouble (x’y’) => (x’y’, Y)
    isSingle x => (x, X)
    failwith (’***ILLEGAL CHAR***’ - ’ in getATOM’);;
getATOM = _ : (string list -> (string # string list))
#getATOM (explode ’( <> b ) ’);

2.4 Parsing \textsc{HOL}

The parser takes atoms from the lexer as input, makes sure that they are
well formed, and builds the ML representation of a term on the fly. Our
first step is to tabulate what we expect the parser to read and what we
want it to return. Table 2.2 overleaf gives a \textsc{BNF} for \textsc{HOL} on the left and
its translation on the right.

The translation schema, \( \mathcal{T} \), is perhaps best explained from the bottom
up.

\textbf{Rule 1} The constant \( T \) translates to \texttt{mk\_CON true}.

\textbf{Rule 2} The constant \( F \) translates to \texttt{mk\_CON false}.

\textbf{Rule 3} A variable, say \( x \), translates to \texttt{mk\_VAR x}.

\textbf{Rule 4} A negation, say \( \sim t \) translates to \texttt{mk\_NEG} whose argument is
the translation of \( t \), e.g. \( \sim F \) translates to \texttt{mk\_NEG(mk\_CON false)}.

\textbf{Rule 5} A parenthesized term, say \( ( t ) \) has the same translation as the
term \( t \) alone.
2.4. **PARSING pHOL**

<table>
<thead>
<tr>
<th>Rule</th>
<th>BNF</th>
<th>Translation schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>atom ::= T</td>
<td>mk_CON true</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>mk_CON false</td>
</tr>
<tr>
<td>3</td>
<td>variable</td>
<td>mk_VAR variable</td>
</tr>
<tr>
<td>4</td>
<td>~ atom</td>
<td>mk_NEG (T atom)</td>
</tr>
<tr>
<td>5</td>
<td>( term )</td>
<td>T term</td>
</tr>
<tr>
<td>6</td>
<td>bin3 ::= atom &amp; bin3</td>
<td>mk_CONJ (T atom) (T bin3)</td>
</tr>
<tr>
<td>7</td>
<td>atom</td>
<td>T atom</td>
</tr>
<tr>
<td>8</td>
<td>bin2 ::= bin3 \lor bin2</td>
<td>mk_DISJ (T bin3) (T bin2)</td>
</tr>
<tr>
<td>9</td>
<td>bin3</td>
<td>T bin3</td>
</tr>
<tr>
<td>10</td>
<td>bin1 ::= bin2 \Rightarrow bin1</td>
<td>mk_IMP (T bin2) (T bin1)</td>
</tr>
<tr>
<td>11</td>
<td>bin2</td>
<td>T bin2</td>
</tr>
<tr>
<td>12</td>
<td>term ::= bin1 \equiv term</td>
<td>mk_EQV (T bin1) (T term)</td>
</tr>
<tr>
<td>13</td>
<td>bin1</td>
<td>T bin1</td>
</tr>
</tbody>
</table>

Table 2.2 BNF for pHOL

Rules 6 and 7, 8 and 9, 10 and 11, 12 and 13 work together in pairs. A single example should suffice. The steps in the translation of the term \( T \ x \land (y \lor F) \) are tabulated below: each transformation arrow is labelled with the rule that is applied for that step.

\[ T \ x \land (y \lor F) \]

\[ \rightarrow mk\_CONJ (T x) (T (y \lor F)) \]

\[ \rightarrow mk\_CONJ (mk\_VAR 'x') (T (y \lor F)) \]

\[ \rightarrow mk\_CONJ (mk\_VAR 'x') (T y \lor F) \]

\[ \rightarrow mk\_CONJ mk\_VAR 'x' (mk\_DISJ (T y) (T F)) \]

\[ \rightarrow mk\_CONJ (mk\_VAR 'x') (mk\_DISJ (mk\_VAR 'y') (T F)) \]

\[ \rightarrow mk\_CONJ (mk\_VAR 'x') (mk\_DISJ (mk\_VAR 'y') (mk\_CON false)) \]

In order to deal with the operators and precedences, we construct a recursive descent parser with separate functions to read off conjunctions, disjunctions, implications, and equivalences. As might be expected from the similarity of their definitions (table 2.2) the code for these functions is much of a muchness. For example, here is suitable code for the function `parseCONJ` which takes responsibility for rules 6 and 7 of table 2.2.

```plaintext
#letrec parseCONJ xX
  = let (x, X) = parseATOM xX in % read leading atom %
    let (y, Y) = getATOM X in % lookahead next atom %
    (y = '/\') \% parse if next = /\ %
    \rightarrow let (z, Z) = parseCONJ Y in
      \(\rightarrow mk\_CONJ x z, Z)\)
    \| (x, X);;
```
The idea being that, when called, `parseCONJ` will look for an input sequence of the form \( A_1 \land A_2 \land \ldots \land A_n \ldots \), where the individual \( A \) terms are atoms. It returns a pair, the parse of the conjunctions, together with what remains to be parsed. It calls `parseATOM` to strip away \( A_1 \). That parse is returned in \( x \), what remains to parse \((\land A_2 \land \ldots \land A_n \ldots)\) in \( X \). If the next atom is anything but an \( \land \), we are done. If the next atom is an \( \land \), we recursively call `parseCONJ` to deal with what lies after it (namely \((A_2 \land \ldots \land A_n \ldots)\)). The parse is returned in \( z \). We then construct the final result \( \text{mk}_\text{CONJ} \, x \, z \) and return it and what is left to parse (represented here by \( \ldots \)) as a pair.

We now define a common template for the `parseCONJ`, `parseDISJ`, `parseIMP`, and `parseEQV`. We need arguments to cover the function analogous to `parseATOM` in `parseCONJ` which strips away the leading term (front), the operator in question (op), the constructor being generated (constructor), and what is left to parse (xx).

```ml
#letrec parseOP/2 front op constructor xX
 = ( let (x, X) = front xX in
   let (y, Y) = getATOM X in
   (y = op) => let (z, Z) = parseOP/2 front op constructor Y in
     (constructor x z, Z)
   | (x, X)
 );;
parseOP/2 =
 : ((string list -> (* # string list)) ->
    string ->
   (* -> * -> *) ->
   string list ->
   (* # string list))
```

It is now a simple matter to define `ParseCONJ`, `ParseDISJ`, etc in terms of `parseOP2`, e.g. `parseCONJ = parseOP2 parseATOM '\land\land' mk_CONJ`.

```ml
#letrec parseATOM xX
 = ( let (x, X) = getATOM xX in
   (x = 'T') => (mk_CON true, X)
   | (x = 'F') => (mk_CON false, X)
   | isLetter x => (mk_VAR x, X)
   | (x = '(') => let (y, Y) = parse X in
     % get expr after ( %
     let (z, Z) = getATOM Y in
     z = ')' => (y, Z) % check for ) %
     | failwith ')' expected'
   | (x = ')') => let (y, Y) = parseATOM X in
     (mk_NEG y, Y)
   | failwith x)
 )
```
The coding of the parser P is now easy. The concatenated string is exploded into a character list, which is then parsed. The parser returns a pair but we require only its first component.


CHAPTER 2. TERMS IN \( \mu \)HOL

EXERCISES 2

Exercise 2.1 Extend the lexer and parser and the abstract datatype \texttt{TERM} to deal with terms involving the quantifiers \( \forall \) and \( \exists \), for example

\[
\neg x . ? y . "(y <> x)"
\]

As with HOL, we use \( \neg \) to represent logical negation, \( ! \) to represent \( \forall \), and \( ? \) to represent \( \exists \).

Exercise 2.2 In the parsing of binary terms, we follow the syntax and accumulate to the right, e.g. we parse \( a \land b \land c \) as \( (a \land (b \land c)) \). Alter the parser to accumulate to the left and return \( (a \land b) \land c \) etc. What is the corresponding syntax? Does the new syntax alter the parse tree in any significant way?
Chapter 3

Theorems in $p$HOL

In this chapter we show you how to: axiomatise a logic, conduct formal proofs in a logic, and how to implement a proof assistant in ML. All the examples in this chapter are for $p$HOL, but techniques extrapolate to HOL itself. This chapter will reinforce your knowledge of ML, show you how to construct and take apart theorems in a sound manner, and how to build more advanced inference rules from the given primitives.

3.1 A formal system for $p$HOL

Before we can carry out reliable proofs in a logic we have to formalise its primitive symbols and operations, the rules of formation for terms, the axioms and the inference rules. The primitive symbols, operations, and formation rules of $p$HOL were described in chapter 2. Axioms are terms that are assumed to hold without proof, e.g. $\vdash T$ and $\vdash \neg F$. Inference rules are the mechanisms that enable us to derive new theorems from the axioms and old theorems. HOL itself (which contains the propositional logic) is founded on 5 axioms and 8 primitive inference rules — everything else is formally derived. Although this makes for a very reliable system, it would be pedagogically counter-productive trying to explain the general principles behind constructing proofs in $p$HOL using this axiomatisation as basis — the axioms and basic inference rules are just too primitive. We follow the style of Manna [74, chapter 2] and take as our basis some theorems and inference rules that are actually derived (rather than assumed) in the HOL system. In this way we raise our level of abstraction to something akin to that of digital logic and can work with more intuitive examples.

A theorem in $p$HOL is represented by a sequent — a pair

\[(\text{hypotheses, conclusion})\]

where hypotheses is a list of assumptions (list of terms) and the conclusion is a term which has been shown to be true under the assumption that each individual hypothesis is true. (Hypotheses are often called “premises” or “assumptions”.) In presentations, we use the turnstile $\vdash$ to denote theoremhood. So
$p, q \vdash p \land q$

displays the theorem $p \land q$ under the assumptions that both $p$ (is true) and $q$ (is true). The hypotheses are listed to the left of the turnstile, the conclusion to the right.

A proof is a tree whose leaves are axioms or established theorems and each of whose internal nodes is a theorem which follows from its immediate ancestors by a rule of inference.

\begin{figure}[h]
\centering
\begin{tikzpicture}
  \node (a) {$\vdash a \sqsupset (b \land c \sqsupset d)$};
  \begin{scope}[every path/.style={-latex}]
    \path (a) edge node [right] {} (b);
    \path (a) edge node [left] {MP} (c);
    \node (b) {$\vdash a$};
    \node (c) {$\vdash (b \land c) \sqsupset d$};
    \node (d) {$\vdash (b \land c)$};
    \node (e) {$\vdash d$};
    \path (c) edge node [right] {} (d);
    \path (d) edge node [left] {MP} (e);
    \path (c) edge node [right] {} (e);
  \end{scope}
\end{tikzpicture}
\caption{Proof tree for $\vdash d$}
\end{figure}

Suppose we have three theorems each with empty hypotheses (the leaves in figure 3.1)

\begin{align*}
  \text{th1} & = \vdash a \sqsupset (b \land c \sqsupset d) \\
  \text{th2} & = \vdash a \\
  \text{th3} & = \vdash b \land c
\end{align*}

and the inference rule MP (modus ponens). MP takes two theorems, the first of the form $G_1 \vdash A \sqsupset B$, the second of the form $G_2 \vdash A$, and from them infers that $G_1 + G_2 \vdash B$. We use the notation $G + H$ to denote the merging of hypotheses\(^1\). Then we can get a proof of $\vdash d$ by two applications of MP. For $MP \text{ th1 th2 } = \vdash b \land c \sqsupset d$ and $MP (MP \text{ th1 th2 }) \text{ th3 } = \vdash d$.

When making formal proofs it is usual to display the steps in the proof in tabular form. Each step should be uniquely labelled, and each proof step should be justified. Here is the formal presentation of our proof of $\vdash d$.

Each step in the table is either given, a theorem or axiom in its own right, or derived from previous lines by application of an inference rule. Thus each step in the table will be a theorem.

\(^1\)For example if $G = \{ a; b \}$ and $H = \{ b; c \}$, then $G + H = \{ a; b; c \}$. There is no point in allowing the same term to appear on the assumption list twice and we make some effort to ensure that the situation does not arise. Likewise we use $G - B$ to denote the \{set\} subtraction of hypotheses. In the case that no member of $B$ is a member of $G$, then $G - B = G$. 

We now present a deduction system for the propositional logic subset of HOL. It contains a number of theorems and inference rules (taken from HOL) which relate to propositional logic. The inference rules allow us to introduce and eliminate assumptions and the operators. These basic theorems and inference rules are presented in a uniform style which we now explain using the inference rule ADD\_ASSUM which permits us to add an assumption to an existing theorem. Here is its presentation:

\[
\text{ADD\_ASSUM: term \rightarrow thm \rightarrow thm} \\
\text{G \downarrow A} \\
\text{\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad B\_G \downarrow A}
\]

from which we learn that ADD\_ASSUM takes two arguments, a term and a theorem, and returns a theorem. The form of the arguments is given on the right, vertically stacked above a horizontal line. When read from top to bottom, the arguments are in the order expected by the theorem or inference rule being specified. Below the horizontal line we find the result of applying the rule in question to these arguments. Terms are written in single quotes for distinction, e.g. 'term', theorems are represented in sequent form.

In this case, if the arguments are a term 'B' and a theorem with hypotheses G and conclusion A, then ADD\_ASSUM constructs a new theorem B\_G \vdash A. The theorem has the same conclusion as the theorem taken as argument, but its list of hypotheses has been enriched by B. For example,

\[
\text{ADD\_ASSUM 'a \land b' ([ a ] \vdash a) = [ a \land b; a ] \vdash a}
\]

Note that in ADD\_ASSUM tm \_thm, if the term \text{tm} is already on the assumption list of the theorem, then the theorem returned is the theorem passed as argument, as in

\[
\text{ADD\_ASSUM 'a \land b' [ a \land b; a ] \vdash a = [ a \land b; a ] \vdash a}
\]
CHAPTER 3. THEOREMS IN $\text{pHOL}$

Basic theorems and inference rules of $\text{pHOL}$ (i)

In the remainder of this chapter we list several axioms and inference rules of $\text{pHOL}$, give examples of their use in inferring new theorems and stronger inference rules, and then describe an implementation of a proof checker for $\text{pHOL}$. We have left out rules for negation since they take too long to explain and add nothing pertinent to our direction. (Find out about them by completing exercise 3.1.) $\text{pHOL}$ theorems and inference rules are given their HOL names and their definitions are precisely their HOL definitions so that little need be unlearned when we move on to HOL in the rest of this text.
3.1. A FORMAL SYSTEM FOR P_HOL

\begin{align*}
\text{DISJ1: } \text{thm} \rightarrow \text{term} \rightarrow \text{thm} & : \quad \text{G} \vdash A \\
& \quad \text{G} \vdash B \\
& \quad \text{------------------------} \\
& \quad \text{G} \vdash A \lor B \\
\text{DISJ2: } \text{term} \rightarrow \text{thm} \rightarrow \text{thm} & : \quad \text{G} \vdash A \\
& \quad \text{G} \vdash B \\
& \quad \text{------------------------} \\
& \quad \text{G} \vdash A \lor B \\
\text{DISJ_CASES: } \text{thm} \rightarrow \text{thm} \rightarrow \text{thm} \rightarrow \text{thm} & : \quad \text{G} \vdash A \lor B \\
& \quad \text{A+G1} \vdash C \\
& \quad \text{B+G2} \vdash C \\
& \quad \text{------------------------} \\
& \quad \text{G+G1+G2} \vdash C \\
\text{DISCH: } \text{term} \rightarrow \text{thm} \rightarrow \text{thm} & : \quad \text{G} \vdash A \\
& \quad \text{G} \vdash B \\
& \quad \text{------------------------} \\
& \quad \text{G-A} \vdash A \rightarrow B \\
\text{UNDISCH: } \text{thm} \rightarrow \text{thm} & : \quad \text{G} \vdash A \rightarrow B \\
& \quad \text{------------------------} \\
& \quad \text{A+G} \vdash B \\
\text{MP: } \text{thm} \rightarrow \text{thm} \rightarrow \text{thm} & : \quad \text{G1} \vdash A \rightarrow B \\
& \quad \text{G2} \vdash A \\
& \quad \text{------------------------} \\
& \quad \text{G1+G2} \vdash B \\
\text{EQV_IMP_RULE: } \text{thm} \rightarrow (\text{thm} \land \text{thm}) & : \quad \text{G} \vdash A \leftrightarrow B \\
& \quad \text{------------------------} \\
& \quad (\text{G} \vdash A \rightarrow B, \text{G} \vdash B \rightarrow A) \\
\text{IMP_ANTISYM_RULE: } \text{thm} \rightarrow \text{thm} \rightarrow \text{thm} & : \quad \text{G1} \vdash A \rightarrow B \\
& \quad \text{G2} \vdash B \rightarrow A \\
& \quad \text{------------------------} \\
& \quad \text{G1+G2} \vdash A \leftrightarrow B
\end{align*}

Basic theorems and inference rules of \textit{p}_HOL (ii)

Axioms for truth and falsity

We take \vdash T and \vdash \neg F as axioms.

Assumptions

Proofs usually start by using \texttt{ASSUME} one or more times to generate one or more leaves in a proof tree. \texttt{ASSUME 'a \land b'} generates the theorem \textit{a \land b} \vdash \textit{a \land b}. 
ADD ASSUM is used to add an extra assumption to an existing theorem. It returns the theorem unchanged if the assumption is already amongst its hypotheses.

PROVE,HYP is used to reduce the number of assumptions in a theorem. If we have th\(4 \vdash a\) and another theorem th\(5\) with \(a\) on its assumption list, then PROVE,HYP th\(4\) th\(5\) will return a new theorem derived from th\(5\) but with \(a\) removed from its assumption list.

### Conjunction

These rules allow us to construct new theorems from existing ones, either by conjoining them, or stripping them down.

CONJ takes two theorems \(G \vdash A\) and \(H \vdash B\) and infers a new theorem with hypotheses \(G + H\) and conclusion \(A \land B\).

CONJUNCT\(1\) is used for stripping away the left term in a conjunction and returning it as a separate theorem.

CONJUNCT\(2\) is used for stripping away the right term in a conjunction and returning it as a separate theorem.

CONJ allows us to prove a larger result in parts and then "join" the parts together. CONJUNCT\(1\) and CONJUNCT\(2\) allow us to use selected portions of a stronger result.

### Disjunction

DISJ1 takes a theorem, say \(th1 = G \vdash x \land y\), and a term, say \(B = \neg x \lor z\), and generates a new theorem in the form of a disjunction with the term on the right. DISJ1 th\(1\) B \(= G \vdash (x \land y) \lor (x \lor z)\).
3.1. A FORMAL SYSTEM FOR $\mathcal{PHOL}$

DISJ2 takes a term and a theorem and generates a new theorem in the form of a disjunction with the term on the left. $\text{DISJ2 } B \text{ th1} = G \vdash (x \lor z) \lor (x \land y)$.

DISJ\_CASES uses a theorem in the form of a disjunction to eliminate assumptions elsewhere. Suppose we have as a theorem $\text{th1} = a \lor b$. Then either $a$ is true or $b$ is true. If we also have as theorems that $\text{th2} = a \vdash c$ and that $\text{th3} = b \vdash c$, it follows that $c$ must also be true. We can generate this theorem by a call $\text{DISJ\_CASES th1 th2 th3} = \vdash c$. The order of the arguments is important — $\text{DISJ\_CASES th1 th3 th2}$ fails.

**THEOREM**

| th1 = \lor x | Given |
| th2 = \lor x \land (y \land z) | DISJ1 \ th1 \ y \land z |
| th3 = \lor (y \land z) \lor x | DISJ2 \ y \land z \ th1 |
| th4 = x, y \lor z | Given |
| th5 = (y \land z) \lor z | Given |
| th6 = y \lor z | DISJ\_CASES th3 th5 th4 |

**Implication**

DISCH is used to extract a specific assumption from the assumption list and make it an explicit part of the conclusion (which takes the form of an implication). The assumption need not be on the assumption list since if we are given $G \vdash B$, then $G \vdash A \lor B$ is a theorem for any term $A$.

UNDISCH works in the opposite direction, stripping away an implicand from a conclusion in the form of an implication and putting it on the assumption list.

MP, modus ponens, takes a theorem whose conclusion is an implication, say $\text{th1} = \vdash (x \lor y \land y \lor z) \lor (x \lor z)$, and a proof of the antecedent $\text{th2} = \vdash x \lor y \land y \lor z$ and allows us to infer that $\vdash x \lor z$ from $\text{MP th1 th2}$.

**THEOREM**

| th1 = c, a \lor b \lor d | Given |
| th2 = c \lor a \lor b \quad \Rightarrow \quad a | DISCH \ a \lor b \ th1 |
| th3 = a \lor b \lor c \quad \Rightarrow \quad a | DISCH \ c \ th1 |
| th4 = c \lor (a \lor b) \quad \Rightarrow \quad c \quad \Rightarrow \quad a | DISCH \ a \lor b \ th3 |
| th5 = a \lor b \lor c \quad \Rightarrow \quad a | UNDISCH \ th4 |
| th6 = d \lor c | Given |
| th7 = a \lor b, d \lor a | MP \ th5 \ th6 |
3.2 Doing proofs in $p$HOL

We present two simple proofs using some of these basic theorems and rules.

Example 3.2.1 Prove that $\vdash a \equiv a$.

The proof first assumes $a$, then discharges $a$ to obtain $\text{th}2 = \vdash a \supset a$. Passing the $\text{th}2$ to $\text{IMP\_ANTISYM\_RULE}$ in both argument positions completes the proof.

Example 3.2.2 Prove $\vdash p \supset (p \lor q)$.

The proof assumes $p$, or's $q$ on the right using $\text{DISJ1}$, and then discharges the assumption.
3.3 A datatype for \( \mu \)HOL theorems

In this section we construct a proof checker for \( \mu \)HOL. The checker implements all the basic theorems and inference rules of section 3.1. The system builds upon the lexer and parser for \( \mu \)HOL terms written in the last chapter. Just as with terms, theorems are implemented as abstract data types.

In order to distinguish arbitrary terms from theorems, we introduce a new type \texttt{THM}. The predefined values of this type are the axioms and the inference rules given in section 3.1. The only primitive functions which yield theorems are the inference rules. The set of values of type \texttt{THM} is just the set of terms which can be derived from the axioms using rules of inference.

Theorems are represented by ( \texttt{TERM list # TERM} ) pairs using the abstract data type for terms and the parser we developed in chapter 2. The mathematical representation \( a \) becomes \( \mu \) \( a \) in \( \mu \)HOL.

Here is a first cut at the definition of \texttt{THM} together with functions \texttt{showL} to show a list of terms and \texttt{showTHM} to show a theorem.

```plaintext
#abstractype THM
  = ( TERM list # TERM )
with
  mk_THM L s = abs_THM (L, s)
  and dest_THM th = rep_THM th;

#letrec showL L = map showTerm L
#let showTHM = showL # showTerm

let th = mk_THM [ mk_VAR 'a'; mk_VAR 'b'; mk_VAR 'c' ];
th = - : THM
```

We can both create theorems and take them apart. For example,
We need several auxiliary functions to ensure that assumptions are not duplicated in assumption lists. \texttt{merge a L} adds a single assumption \texttt{a} to an assumption list \texttt{L} if it is not there already. \texttt{Lmerge X Y} merges the assumptions of \texttt{X} into those of \texttt{Y} one at a time. \texttt{remove a L} removes all occurrences of \texttt{a} from \texttt{L} (we play it safe!).

\begin{verbatim}
#letrec merge (a:TERM) L = (mem a L =/> L /| a /: L)
and Lmerge (X:TERM list) Y = (Lmerge (tl X) (merge (hd X) Y)) ? Y
and remove (asm:TERM) L = filter (\ x . not (asm = x)) L;;
merge = - : (TERM -> TERM list -> TERM list)
Lmerge = - : (TERM list -> TERM list -> TERM list)
remove = - : (TERM -> TERM list -> TERM list)
\end{verbatim}

We now show you how to write the axioms and built-in inference rules.

\textbf{Axioms.} The axioms are trivial.

\begin{verbatim}
#let TRUTH = mk_THM [] (mk_CON true)
and FALSITY = mk_THM [] (mk_NEG(mk_CON false));;
TRUTH = - : THM
FALSITY = - : THM
\end{verbatim}

\textbf{ASSUME} takes a sentence \texttt{s} to a theorem of the form \texttt{s |- s}.

\begin{verbatim}
#let ASSUME t = mk_THM [ t ] t;;
ASSUME = - : (TERM -> THM)
#showTHM (ASSUME (P 'b /	extasciicircum 'c));
(['b /	extasciicircum 'c'], ['b /	extasciicircum 'c']) : (string list # string)
\end{verbatim}

In \texttt{ASSUME} we simply make a new theorem with the term on the assumption list and as the conclusion.

\textbf{ADD_ASSUM} is not much more complicated.
The arguments to \texttt{ADD\_ASSUM} are the extra assumption passed as a term and a theorem. We split the theorem into its hypothesis and conclusion. The new theorem is constructed merely by merging \texttt{t} into the assumption list and taking the same conclusion.

\textbf{MP.}

Having taken apart the arguments to \texttt{MP}, we then have to make sure that the conclusion of \texttt{th1} is an implication (\texttt{dest\_imp} will do this) and that the antecedent of this theorem (\texttt{lhs}) is identical to the conclusion of \texttt{th2}. If so we can generate a new theorem merging the two lists of hypotheses and taking the implicand of \texttt{th2} (\texttt{rhs}) as the conclusion. If the first argument \texttt{th2} is not an implication, then \texttt{dest\_IMP} will fail. Any internal message is overridden by our outermost use of \texttt{failwith}.

We carry on in this vein to construct all the built-in axioms and rules. Note that we must be very careful when using \texttt{mk\_THM}. Although the strong type checking of ML will prevent many errors of a trivial kind and disallow such things as \texttt{MP 'a == b' a'} on the grounds that the arguments
are strings not theorems, we can still make slips and produce theorems or inference rules which are unsound. For example

```haskell
#let NONSENSE = mk.THM [] (mk.COM false);;
NONSENSE = - : THM

#showTHM NONSENSE;;
([], `|- F`) : (string list # string)
```

Or we could have written the penultimate line of MP as (mk.THM hyp1 rhs) by mistake. It pays therefore to have a very small set of primitives that can be easily checked by visual inspection, exhaustively tested, or proven correct, and find ways of constructing higher level tools from them in a sound manner.

But allowing mk.THM at all is a major weakness. It is accessible to all users and someone is bound to misuse it. This is not good enough if we are serious about producing reliable proofs. Instead we write the whole definition as an abstract data type and place all the primitive axioms and rules inside the body of the abstract data type, taking care not to define mk.THM at all but to use the primitive abs.THM directly. Now since there is no mk.THM and there is no access to abs.THM outside the body of the datatype, users of the system cannot fake theorems. The only way they can be constructed is through an inference rule.

Here is a complete definition of this data type in this style.
Abstract datatype for theorems (i)
and DISJ_CASES th1 th2 th3
= ( let (hyp1, concl1) = rep_THM th1
   and (hyp2, concl2) = rep_THM th2
   and (hyp3, concl3) = rep_THM th3
   in
   let (lhs, rhs) = dest_DISJ concl1 in
   (concl2 = concl3)
   & (mem lhs hyp2)
   & (mem rhs hyp3)
   => abs_THM (Lmerge hyp1
                 (Lmerge (remove lhs hyp2)
                 (remove rhs hyp3)),
               concl2)
   | fail
   )? failwith 'DISJ_CASES'

and DISJ t th
= ( let (hyp, concl) = rep_THM th in
    abs_THM (hyp, mk_DISJ t concl)
   )? failwith 'DISJ'

and DISJ2 t th
= ( let (hyp, concl) = rep_THM th in
    abs_THM (hyp, mk_DISJ concl t)
   )? failwith 'DISJ2'

and DISCH t th1
= ( let (hyp1, concl1) = rep_THM th1 in
    abs_THM (remove t hyp1, mk_IMP t concl1)
   )? failwith 'DISCH'

and UNDISCH th1
= ( let (hyp1, concl1) = rep_THM th1 in
    let (lhs, rhs) = dest_IMP concl1 in
    abs_THM (merge lhs hyp1, concl1)
   )? failwith 'UNDISCH'

and MP th1 th2
= ( let (hyp1, concl1) = rep_THM th1
    and (hyp2, concl2) = rep_THM th2 in
    let (lhs, rhs) = dest_IMP concl1 in
    (lhs = concl2) => abs_THM (Lmerge hyp1 hyp2, rhs)
   | fail
   )? failwith 'MP'

and EQ_IMP_RULE th1
= ( let (hyp1, concl1) = rep_THM th1 in
    let (lhs, rhs) = dest_EQV concl1 in
    (abs_THM (hyp1, mk_IMP lhs rhs), abs_THM (hyp1, mk_IMP rhs lhs))
   )? failwith 'EQ_IMP_RULE'

Abstract datatype for theorems (ii)
3.4 Machine checked proofs in \( \mu \text{HOL} \)

We now use our proof checker to prove two simple theorems.

**Example 3.4.3** Prove \( \vdash a \supset (b \supset (a \land b)) \)

We start by assuming \( a \) and \( b \) separately and then conjoining them together to form \( \text{th3} = a, b \vdash a \land b \).
All that is left is to discharge the assumptions in the appropriate order.

Example 3.4.4 Prove \( a \land b \equiv b \land a \).

This proof is a little larger than the previous one and needs a strategy. As is typical for equivalence proofs, we split the problem up and prove implication both ways. We get the result we want by combining the two parts of the proof using \texttt{IMPANTISYM RULE}. The proof is interesting in that the two implication proofs have identical structures. It is a simple matter to write an ML function powerful enough to handle both implication proofs.

1. **Show** \( a \land b \supset b \land a \). \( \texttt{th1} \) simply assumes \( a \land b \). We can obtain two more theorems \( \texttt{th2} = a \land b \vdash a \) and \( \texttt{th2} = a \land b \vdash b \) from \( \texttt{th1} \) using \texttt{CONJUNCT1} and \texttt{CONJUNCT2} respectively, and conjoin them in “reverse order”. The last step is to discharge the one assumption. Here is the proof in \( p\text{HOL} \):

\[
\begin{array}{ll}
\text{THEOREM} & \text{JUSTIFICATION} \\
\texttt{th1} = a \land b \vdash a \land b & \text{let th1 = \texttt{ASSUME 'a /\ b'}} \\
\texttt{th2} = a \land b \vdash a & \text{let th2 = \texttt{CONJUNCT1 th1}} \\
\texttt{th3} = a \land b \vdash b & \text{let th3 = \texttt{CONJUNCT2 th1}} \\
\texttt{th4} = a \land b \vdash b \land a & \text{let th4 = \texttt{CONJ th3 th2}} \\
\texttt{th5} = \vdash a \land b \Rightarrow b \land a & \text{let th5 = \texttt{DISCH 'a /\ b'}} \\
\end{array}
\]
2. **Show** \( b \land a \supset a \land b \). It is obvious that the second part of the proof will be very similar to the first part. We now do with \( b \land a \) what we just did with \( a \land b \). Instead of more or less duplicating the proof steps we show the power of embedding a logic in a powerful programming language by writing a function to carry out these proof steps:

```ocaml
#let f s
  = let s' = P s
     in
     let th1 = ASSUME s'
     in
     DISCH s' (CUNJ (CUNJUNCT2 th1) (CUNJUNCT1 th1));
     f = - : (string -> THM)
#showTHM (f `a \land b`);;
(\[\]`|/- (a \land b) \implies (b \land a)`\] : (string list # string)

#showTHM (f `b \land a`);;
(\[\]`|/- (b \land a) \implies (a \land b)`\] : (string list # string)

#showTHM (f `a \land c` \land `((e \implies f) \lor g)`);;
(\[\]`|/- (((a \land c) \land ((e \implies f) \lor g)) \implies (((e \implies f) \lor g) \land (a \land c))`\] : (string list # string)

The approach is sound, for \(\text{CUNJUNCT1 th1} \) and \(\text{CUNJUNCT2 th2} \) will both fail if \(\text{th1} \) is not a theorem in the form of a conjunction. The proof of the theorem is now simply:

```ocaml
#let th = IMP_ANTISYM_RULE (f `a \land b`) (f `b \land a`);
th = - : THM
#showTHM th; ;
(\[\]`|/- ((a \land b) \iff (b \land a))`\] : (string list # string)

#let th = IMP_ANTISYM_RULE (f `a \land b`) (f `b \land a`)
  where f s
    = let s' = P s
     in
     let th1 = ASSUME s'
     in
     DISCH s' (CUNJ (CUNJUNCT2 th1) (CUNJUNCT1 th1));
  th = - : THM
#showTHM th; ;
(\[\]`|/- ((a \land b) \iff (b \land a))`\] : (string list # string)
```

We have given two versions of the same theorem. The second uses an ML clause to make a local copy of the auxiliary theorem \( f \ s \). We think that this is good practice when the auxiliary theorem is of little general utility.
3.5 Summary

Note two drawbacks with our \( p \text{-HOL} \) implementation: ML will echo back the names and types of the theorems we prove, and we have to explicitly display our theorems. Our little system requires an interface to show the results as we go (intercepting ML responses like \( \text{th} = - : \text{THM} \) and showing the actual theorem instead and a library mechanism to save theorems. (This is not set as an exercise — it would be quite an effort for even the most the enthusiastic reader.)

At this stage we should take a look at the issue of the soundness of our system. We have defined a core system containing some 16 axioms and inference rules. Their implementations must be correct (without even the slightest loophole), since proofs in \( p \text{-HOL} \) depend upon them. One way to improve things would be to implement a core set of givens (finding these is an interesting exercise) in the data type, and express everything else in terms of these but outside the abstract data type. For example, if \text{dest\_THM}, \text{MP}, \text{DISCH} are givens then \text{PROVE\_HYP} becomes:

\[
\begin{align*}
\text{let PROVE\_HYP th1 th2 = let (hyp1, concl1) = dest\_THM th1 in MP (DISCH concl1 th2) th1 ;;}; \\
\text{PROVE\_HYP = - : (THM \to THM \to THM)}
\end{align*}
\]

Several behind-the-scenes functions are used heavily. These include the \text{isSame} and \text{subst} functions over terms; the \text{isIn}, \text{merge}, \text{Lmerge}, and \text{remove} operations on lists; not forgetting the soundness of the ML type inference scheme and the correctness of the ML compiler. It is very unlikely that all these items are indeed correct, but with heavy use over a number of years most of the bugs get wrinkled out and proof checkers do become reliable. That is not to say they can be completely trusted. There is no way round this dilemma. As professionals we must be honest and claim that our proofs are probably correct but subject to some doubt. The best way round the impasse may be to follow a suggestion by Malcolm Newey and use proof checkers such as ours not only to generate proofs but also to generate “assembly language” versions of the proof in terms of a minimum number of inference rules and auxiliary theorems. The assembly language version of the proof would then be run again on a minimal proof checker, so small that there would be a chance of verifying its correctness. Unfortunately the regression does not stop there; was the hardware correct, etc...
3.5. SUMMARY

EXERCISES 3

Exercise 3.1  Explain and then implement Manna’s rules [74, page110] for negation:

**NOT_INTRO1: term → thm → thm → thm**

\[ \begin{array}{l}
\text{\textquoteleft} A \text{\textquoteleft} \\
G \vdash A \vdash B \\
G \vdash A \vdash \neg B \\
\hline 
G \vdash \neg A \\
\end{array} \]

**NOT_ELIM1: term → thm → thm → thm**

\[ \begin{array}{l}
\text{\textquoteleft} A \text{\textquoteleft} \\
G \vdash B \\
G \vdash \neg B \\
\hline 
G \vdash A \\
\end{array} \]

**NOT_INTRO2: thm → thm**

\[ \begin{array}{l}
G \vdash A \\
\hline 
G \vdash \neg A \\
\end{array} \]

**NOT_ELIM2: thm → thm**

\[ \begin{array}{l}
G \vdash \neg A \\
\hline 
G \vdash A \\
\end{array} \]

Exercise 3.2  What inference rules are implemented by the following functions:

```ml
let NOT_INTRO thm
  = ( let (hyp, concl) = dest_THM thm in
    let (lhs, rhs) = dest_IMP concl in
    isSome rhs (mk_IMP false)
    => abs_THM (hyp, mk_NEG lhs)
    | failwith 'NEG_INTRO'
    ) ? failwith 'NEG_INTRO';;

let NOT_ELIM thm
  = ( let (hyp, concl) = rep_THM thm in
    let rhs = dest_NEG concl in
    abs_THM (hyp, mk_IMP rhs (mk_IMP false))
    ) ? failwith 'NEG_ELIM';;
```
Why the enveloping calls on `failwith`?

**Exercise 3.3** Prove or disprove \( \vdash (p \land q) \supset (r \supset (p \land q)) \).

**Exercise 3.4** Prove or disprove \( \vdash (a \supset (b \supset c)) \supset ((a \supset b) \supset (a \supset c)) \).

**Exercise 3.5** Write `DISCH_LIST : THM \rightarrow THM`, a derived inference rule which discharges all the hypotheses from a theorem.

**Exercise 3.6** Write a derived inference rule `SYM : THM \rightarrow THM` which takes a theorem of the form \( G \vdash A \equiv B \) and returns the theorem \( G \vdash B \equiv A \).

**Exercise 3.7** How would you change the sum type within `THM` to accommodate the quantifiers `\forall` and `\exists`. Locate and implement inference rules for both these quantifiers.

**Exercise 3.8** Doing proofs in \( L\) HOL is quite tricky and rather tedious after a while. The logic is so simple that proofs can be automated. Write a tautology checker for \( L\) HOL that takes a string and returns `true` if it is a tautology. Here is a brute force algorithm:

1. parse the string into a term, say \( t \), and extract from it all distinct variable names.

2. each distinct variable may take on either the value `true` or the value `false` independently. Whatever the combination, \( t \) will evaluate to `true` iff it is a tautology. We construct from \( t \) a list containing all possible substitutions for the variables.

3. we now have a (long) list of terms whose leaves are constants. Write a function `eval` to navigate a single such term and returns its value.

4. finally we map `eval` down the list of constant terms and check that each evaluation returns `true`.

Using `isTT` as a model, write a new inference rule `TT : TERM \rightarrow THM` which makes a theorem from its argument should it be a tautology, and otherwise fails. Would you trust theorems generated in this manner?
Part II

HOL
Chapter 4

The HOL notation

This chapter serves as an informal introduction to the HOL notation. A complete formal definition of the syntax and semantics of HOL is given [106]. HOL is a strongly typed logic based on the λ calculus. It includes propositional calculus and first order logic and adds the capability of defining user definable functions, the ability to pass functions as arguments to other functions and for functions to return functions as results. It is an expressive logic: instead of talking about individuals we can talk about sets, functions, groups, etc. directly.

In this chapter, we remind you of first order logic, introduce higher order features through examples, and discuss type constraints in HOL. We then give the HOL inference rules for the quantifiers ∀ and ∃ and prove one very useful theorem, not expressible in first order logic, variants of which are heavily used in hardware proofs. We close with an example of a backward proof, the style in which our hardware proofs will be conducted.

4.1 The HOL first order subset

First order logic extends propositional logic (section 2.1) by allowing quantification, conditionals and more types, e.g. the natural numbers, pairs, and lists. The extensions are:

Numeric constants, variables and operators are allowed in logical statements, but not as logical statements since the latter must be either true or false.

- the natural numbers are the non-negative integers, 0, 1, 2, ...
- numeric variables (which take on natural number values only)
- the successor and predecessor functions are built-in. Their definitions are:

\[
\begin{align*}
\text{SUC} \ n & = n + 1 \\
\text{PRED} \ 0 & = 0 \\
\text{PRED} \ (n+1) & = n
\end{align*}
\]
Since the predecessor of 0 is 0, \( \text{PRED} (\text{SUC } n) = n \) is true for all numbers \( n \), but \( \text{SUC}(\text{PRED } n) = n \) is only true for \( n > 0 \).

- several infix numeric operators are built in: +, -, \( \times \), DIV, MOD and EXP. As examples, \( 14 \text{ DIV } 4 \) equals 3, \( 14 \text{ MOD } 4 \) equals 2, \( 2 \text{ EXP } 4 \) represents the mathematical \( 2^4 \) (or 16). Since negative values are not allowed, we do subtraction by cases giving a special clause if either argument is zero \((0 - 3 = 0 \text{ with our definition})\). If neither argument is zero, then both can be decremented and we make a recursive call to the definition.

\[
\begin{align*}
\text{SUB } 0 \ b &= 0 & a = 0 \\
\text{SUB } a \ 0 &= a & b = 0 \\
\text{SUB } a \ b &= \text{SUB} \ (\text{PRED } a) \ (\text{PRED } b) & a, b > 0
\end{align*}
\]

- HOL also provides a set of infix comparators. = is polymorphic and may be used to compare any two arguments of the same type. <, >, \( \leq \), \( \geq \) operate only on numeric arguments. For example, \( (a < 5 \land b < 5) \lor (a \times b = 25) \) is a false statement.

- the boolean and numeric operators and the comparators are given precedences.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXP</td>
<td>9</td>
</tr>
<tr>
<td>( \times ), DIV, MOD</td>
<td>8</td>
</tr>
<tr>
<td>+, -</td>
<td>7</td>
</tr>
<tr>
<td>=, &gt;, ( \leq ), ( \geq )</td>
<td>6</td>
</tr>
<tr>
<td>( \sim )</td>
<td>5</td>
</tr>
<tr>
<td>( \land )</td>
<td>4</td>
</tr>
<tr>
<td>( \lor )</td>
<td>3</td>
</tr>
<tr>
<td>( \square )</td>
<td>2</td>
</tr>
<tr>
<td>( \equiv )</td>
<td>1</td>
</tr>
</tbody>
</table>

**Table 4.1** Operator precedences

In order: EXP binds tightest, then \( \times \), DIV, MOD which are equally ranked, then +, -, then the comparators (which are also equally ranked), and finally the logical operators, one by one. Table 4.1 tabulates these precedences. Of course, sub-expressions in parentheses are evaluated first.
4.2. **HIGHER ORDER LOGIC**

Tuples, lists and conditionals appear in HOL exactly as they did in ML. The built-in pair operators `FST`, `SND` and list operators `HD`, `TL`, `CONS` are written in upper-case. The empty list is written `[]` and the conditional expression `a => b | c`.

The binders `∀`, `∃` introduce new variables and delimit their scope. Note that the body associated with a binder is a logical expression.

- The universal quantifier is written `∀` and read *for all*. `∀ a . expr` means *for all* `a it is the case that* `expr holds*, e.g.
  \[
  ∀ a b . (a \equiv b) ∨ (a ⊑ b) ∧ (b ⊑ a)
  \]
  states that "for all values of `a` and `b`, `a` being equivalent to `b` implies that both `a` implies `b` and `b` implies `a", and

  \[
  ∀ n . \text{PRED} (\text{SUC} n) = n, \text{and}
  \]
  \[
  ∀ n . (n > 0) ⊃ (\text{SUC} (\text{PRED} n) = n)
  \]
  express the two facts about `SUC` and `PRED` mentioned above. We will be using the universal quantifier in our correctness statements about hardware components to express the fact that they hold true over all possible input and output sequences.

- The existential quantifier is written `∃` and read *there exists*. `∃ a . expr` means *for some* `a (at least one) it is the case that* `expr holds*, e.g.
  \[
  ∀ a . ∃ b . b = \text{SUC} a
  \]
  states that for every number `a`, there is a number `b` which equals the successor of `a`, i.e. every number has a successor. This statement is true. On the other hand `∀ a . ∃ b . b < a` is false in the natural numbers since there is no natural number less than zero. We will be using the existential quantifier in our definitions of hardware components to single out and name purely internal (hidden) wires.

**4.2 Higher order logic**

Higher order logic is based upon the typed λ-calculus and so allows variables to range over functions. Functions may be arguments to functions, and functions may be returned as the results of other expressions.

**λ-calculus.** You should have (at least) a nodding acquaintance with the λ-calculus which is the formal basis for HOL. You should:
• know the scope rules for bound and for free variables, e.g. that in

\[ \lambda x \cdot x \times ((\lambda z \cdot z) \cdot y) + z \]

\(x\) is bound, \(y\) is free, and \(z\) is both bound (the first and second occurrences) and free (the third occurrence).

• know how to define a function using \(\lambda\). E.g. if \(\text{suc } n = n + 1\) then we can write \(\text{suc } = \lambda n \cdot n + 1\), and if \(\text{mult } x y = x \times y\) then \(\text{mult } = \lambda x \cdot \lambda y \cdot x \times y\) or \(\lambda x y \cdot x \times y\) for short.

• be familiar with \(\alpha\) and \(\beta\) conversion. Intuitively, \(\alpha\) conversion allows the systematic renaming of variables, e.g. \(\lambda x \cdot x =_\alpha \lambda y \cdot y\) and \(\beta\) conversion is the function call mechanism, that is

\[ (\lambda x \cdot \text{body}) \; \text{arg} \equiv_\beta [\text{arg} / x] \; \text{body} \]

where \([\text{arg} / x] \; \text{body}\) is read as “substitute arg for each free occurrence of \(x\) within body”.

Importantly, function definitions may be passed as arguments and when they are, they are handled in exactly the same way as constants and variables. For example, here are definitions of \(\text{if}\), \(\text{true}\), and \(\text{false}\) which work together:

\[
\begin{align*}
\text{if } &= \lambda b x y \cdot b x y \\
\text{true } &= \lambda x y \cdot x \\
\text{false } &= \lambda x y \cdot y
\end{align*}
\]

As one might expect, the expression \(\text{if true A else B}\) reduces to \(A\).

\[
\begin{align*}
\text{if true then A else B} &= (\lambda b x y \cdot b x y) \; \text{true A B} \quad \text{defn. of if} \\
\rightarrow_\beta &= (\lambda x y \cdot \text{true x y}) \; A \; B \quad \text{\(\beta\) reduction} \\
\rightarrow_\beta &= (\lambda y \cdot \text{true A y}) \; B \quad \text{\(\beta\) reduction} \\
\rightarrow_\beta &= \text{true A B} \quad \text{\(\beta\) reduction} \\
\rightarrow_\beta &= (\lambda x y \cdot x) \; A \; B \quad \text{defn. of true} \\
\rightarrow_\beta &= (\lambda y \cdot A) \; B \quad \text{\(\beta\) reduction} \\
\rightarrow_\beta &= A \quad \text{\(\beta\) reduction}
\end{align*}
\]

• care is required to avoid the “name capture problem”. In the example below, the intent is to add together an external (global) \(y\) value and \(3\). Proceeding with the rules stated so far
\[(\lambda x \cdot \lambda y \cdot x + y) \ y \ 3\]
\[\not\to_\beta \ ([y/x] (\lambda y \cdot x + y)) \ 3\]
\[= (\lambda y \cdot y + y) \ 3\]
\[= 3 + 3\]

—which is not what was intended. The way round this problem is to check for the possibility of name capture before we attempt a \(\beta\) conversion. Any offending bound variable and its occurrences in the body are renamed before we actually carry out the \(\beta\) conversion, as below

\[(\lambda x \cdot \lambda y \cdot x \times x + y \times y) \ y \ 3\]
\[-\to_\alpha (\lambda x \cdot \lambda z \cdot x \times x + z \times z) \ y \ 3 \quad \text{\(\alpha\) conversion}\]
\[-\to_\beta (\lambda z \cdot y \times y + z \times z) \ 3 \quad \text{\(\beta\) conversion}\]
\[-\to_\beta y \times y + 3 \times 3 \quad \text{\(\beta\) conversion}\]

How will we make use of \(\lambda\)? Here is an example taken from signal propagation relations in a row of \(n+1\) unit delays.

![Line of unit delays](image)

Figure 4.1 Line of unit delays

Emerging from a block of \(n\) unit delays we have a signal relation \(q(t+n) = i \ t\) which states that the input signal at time \(t\) will emerge \(n\) clock ticks later on \(q\). This signal then goes through the rightmost delay and emerges one clock tick later on the line \(z\). This gives another constraint on the behaviour of \(q\) namely that \(z(t+1) = q \ t\). Since the signal on \(q\) must be the same however we look at it, we have to solve these simultaneous relations for \(q\).

\[q(t+n) = i \ t \quad (i)\]
\[z(t+1) = q \ t \quad (ii)\]

Reversing \((ii)\) we get \(q \ t = z(t+1)\). Using the \(\lambda\) notation, we can rewrite this as \(q = \lambda t \cdot z(t+1)\). If we now substitute this definition for \(q\) into \((i)\) we get \((\lambda t \cdot z(t+1)) (t+n) = i \ t\) which can be \(\beta\) reduced to our desired result, \(z(t+n+1) = i \ t\).
**Recommended texts.** For practical introductions to the lambda calculus, try [37, 45, 64]. For in-depth reading, try [3, 58, 92].

**Higher order functions.** A function is first order if none of its arguments is a function, i.e. all of its arguments are data. A second order function allows one or several of its arguments to be first order functions, and so on. We call any function of order higher than first a higher order function. Typical examples are furnished by map and filter. Higher order functions arise naturally when we seek out and formalise common patterns of function definition. This encourages definitions (and specifications) to be written at a higher level of abstraction, thus encouraging clarity and succinctness.

Higher order properties are essential when we come to describe properties of time-varying signals. For example, we define the rise and fall of a clock $clk$ at time $t$ by

$$
\forall \text{ clk } t . \text{ rise clk } t = \sim(\text{ clk } t) \land (\text{ clk}(t+1))
$$

$$
\forall \text{ clk } t . \text{ fall clk } t = (\text{ clk } t) \land \sim(\text{ clk}(t+1))
$$

and for a signal $\text{ sig}$ to be stable at value $v$ at and between times $t_1$ and $t_2$ by

$$
\forall \text{ sig } v t_1 t_2 . \text{ stable sig } v t_1 t_2 \\
= \forall t . (t_1 \leq t \land t \leq t_2) \sqcap (\text{ sig } t = v)
$$

**Extensionality.** The principle of extensionality, PE, states that if two functions return the same result for all values of an argument, then they are the same function. We may write this as

$$
\text{ PE } = (\forall x . (f x = g x)) \sqcap (f = g)
$$

Suppose we define our own curried version, $\\text{ ADD x y } = x + y$, of the $+$ operation, $\text{ ADD a 6}$ and $\text{ ADD 3 \ (ADD 5 6)}$ are typical calls on $\text{ ADD}$. Currying has the advantage that it allows partial applications. Since $\forall x . \text{ INC x} = \text{ ADD 1 x}$ we are permitted to define $\text{ INC } = \text{ ADD 1}$. INC is a function requiring one argument, a num, and returns a num value, e.g. $\text{ INC 5}$ has value $\text{ ADD 1 5 = 6}$.

### 4.3 Types

HOL is a **typed** logic. Besides being well-formed (syntactically correct), all expressions in HOL are automatically checked for consistent and meaningful usage. There are two built-in or ground types: bool for logical quantities and num for numeric quantities. More complicated types may be constructed from these ground types by pairing, constructing lists, and by making function definitions.
4.3. Types

Constants. The type of a constant is manifest: T and F are both of type bool; 0, 1, 2, ... are all of type num.

First order variables. Each HOL variable is given a type, and may be replaced only by values of that given type in any one context. The type of a variable can nearly always be inferred from its usage. For example, in the term \((a \supset b) \land (b \supset c) \supset (a \supset c)\), a, b, and c must all be of type bool since they are used as arguments to logical operations which require bool arguments. Again in, \((a \times b > 0) \supset (a > 0 \land b > 0)\), a and b must both be of type num since they are used as arguments to multiply and, consistent with that, to the comparator \(\supset\). The HOL type checker rejects such expressions as \(5 \supset T\) since we cannot compare a num with a bool; and \((a = 5) \land (a \equiv b)\) — since, to be consistent, a should be a num in the left conjunct and a bool in the right conjunct. The type of b is bool since it occurs as an argument to \(\equiv\).

Adding type information. There are cases when the HOL system cannot infer what type a variable has. For example we cannot tell from \((a = b) \equiv (b = a)\) whether a and b are both bool or both num (or both whatever). We do know that they are of the same type, but not what that type is. In cases like this, the HOL system will ask you to annotate the expression with some type information. Type information is added by appending either \(:\) bool or \(:\) num to one instance of the variable in the expression. Either

\[
(a:\text{num} = b:\text{num}) \equiv (b = a) \quad \text{or just} \quad (a:\text{num} = b) \equiv (b = a)
\]

will do the job since HOL will infer that the type of b is num since it is being compared to the num variable a. In this case, it is not necessary to supply any type information for b. However, if you do supply extra type information, HOL will use it and check to see that you got it right.

Pairs. If a is of type \(\alpha\) and b is of type \(\beta\), then the pair \((a, b)\) has the type \(\alpha \# \beta\). Thus \((4, T)\) has the type \(\text{num} \# \text{bool}\) and \((4, (T, F))\) has the type \((\text{num} \# (\text{bool} \# \text{bool}))\).

Lists. Elements of the same list must have the same type. The type of the list \([T; F; T]\) is bool list. \([1, F); (2, T); (3, T)\] has the type \((\text{num} \# \text{bool})\) list, and \([1; 2]; []; [3; 4; 5]\) is a \text{num list list}.

Functions with one argument. If \(f\) is a function which takes a single argument of type \(\alpha\) and returns a value of type \(\beta\) then the type of \(f\) is written \((\alpha \rightarrow \beta)\). For example, the type of \text{SUC} is \(\text{num} \rightarrow \text{num}\) and the type of \text{echoecho}\( x = (x, x+x)\) is \(\text{num} \rightarrow (\text{num} \# \text{num})\). The function \text{ADD} defined by \text{ADD}(x, y) = x + y has type \((\text{num} \# \text{num}) \rightarrow \text{num}\), i.e. it takes a single argument, which is a pair of nums, and returns a num. We
deduce the type of an argument from its use within the body of the function. As a final example, in

$$\text{PMI} = \forall P . (P \, 0) \land (\forall n . P \, n \supset P(n+1)) \supset \forall n . P \, n$$

we may infer that \(n\) is a \texttt{num} since it is an operand to an addition, and that \(P\) is a predicate with one \texttt{num} argument since \(P \, n\) occurs in the body of universal quantification which is itself a consequent. Thus the type of \(P\) is \texttt{num \rightarrow bool}.

... and with two arguments or more. If \(f\) is a curried function which takes two arguments of types \(\alpha\) and \(\beta\) respectively and returns a value of type \(\gamma\) then the type of \(f\) is written \((\alpha \rightarrow (\beta \rightarrow \gamma))\). The \(\rightarrow\) operator is right associative, so that we may drop the pair of inner parentheses and write \(\alpha \rightarrow \beta \rightarrow \gamma\). E.g. the type of \texttt{ADD} defined by \texttt{ADD \ x \ y = x + y} is written \texttt{num \rightarrow num \rightarrow num} and the type of \texttt{LESS} where \texttt{LESS \ a \ b = a < b} is written \texttt{num \rightarrow num \rightarrow bool}. The notation extends in an obvious fashion to functions with three or more arguments.

\textit{\lambda expressions.} How to type lambda expressions should now be obvious. The type of the bound variable is deduced from its use within the body of the lambda expression. If the type of \texttt{body} is \(\beta\) and the type of \texttt{arg} \(\alpha\), then the type of \(\lambda \texttt{arg} . \texttt{body}\) is \(\alpha \rightarrow \beta\). As an example, the type of \(\lambda x . x + 1\) is \texttt{num \rightarrow num}, and the type of \(\lambda (x, y) . (x = 0 \land y = 0)\) is \texttt{(num \# num) \rightarrow bool}.

\textit{Polymorphism.} \(=\) is polymorphic — it accepts two arguments of the same type and returns a \texttt{bool} value. Thus \(a = (b \land c)\) and \(5 = d\) and are both legal statements. The type of \(=\) is written \(\ast \rightarrow \ast \rightarrow \texttt{bool}\). In \(5 = d\), the type checker matches the type of \(5\), namely \texttt{num}, against \(\ast\) and then expects the second argument (\(d\)) also to be of type \texttt{num}. As a second example, the conditional is implemented as a function \texttt{COND} with three arguments. The first must be of type \texttt{bool}. The second and third arguments must be of the same type, but are otherwise unconstrained. The type of the result is this same type. Thus the type of \texttt{COND} is written \texttt{bool \rightarrow \ast \rightarrow \ast \rightarrow \ast}.

\texttt{FST} and \texttt{SND} operate on pairs and return either the first or the second argument respectively. The components of a pair are freely chosen and may be of distinct types, thus we denote the type of an arbitrary pair not by \((\ast \# \ast)\) which would imply that both components of the pair were of the same type, but by \((\ast \# \ast\ast)\). (We just keep adding asterisks when we need new polymorphic types.) Thus we type \texttt{FST} as \((\ast \# \ast\ast) \rightarrow \ast\) and \texttt{SND} as \((\ast \# \ast\ast) \rightarrow \ast\ast\).
Several list operators are built-in and they all operate on any type of list. Whatever the type of the list argument to \texttt{TL} it always returns a list of the same type. We write the type of \texttt{TL} as \texttt{* list \rightarrow * list}. Whatever the type of list argument to \texttt{HD} it always returns an element of the same type. We write the type of \texttt{HD} as \texttt{* list \rightarrow *}. The type of \texttt{CONS} comes to be \texttt{* \rightarrow * list \rightarrow * list} which ensures that when a new item is cons’ed onto the front of a list, it has the same type as the rest of the list.

Operators. The built-in binary logical, numeric and comparison operators are infix for ease of use. \texttt{\sim} and the binders, are of course, unary operators. Their types are tabulated below:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{\sim}</td>
<td>\texttt{bool \rightarrow bool}</td>
</tr>
<tr>
<td>\texttt{&amp;, \forall, \exists, \equiv}</td>
<td>\texttt{bool \rightarrow bool \rightarrow bool}</td>
</tr>
<tr>
<td>\texttt{+, -, *, DIV, MOD, EXP}</td>
<td>\texttt{num \rightarrow num \rightarrow num}</td>
</tr>
<tr>
<td>\texttt{&lt;, &gt;, &lt;\leq, &gt;\geq}</td>
<td>\texttt{num \rightarrow num \rightarrow bool}</td>
</tr>
<tr>
<td>\texttt{=}</td>
<td>\texttt{* \rightarrow * \rightarrow bool}</td>
</tr>
<tr>
<td>\texttt{\forall, \exists}</td>
<td>\texttt{* \rightarrow bool}</td>
</tr>
<tr>
<td>\texttt{\lambda}</td>
<td>\texttt{* \rightarrow **}</td>
</tr>
</tbody>
</table>

4.4 Other background material

We assume exposure to a regular computer science, computer engineering, or electrical engineering background and expect you to have taken at least one course in digital design and be familiar with the basics of primitive recursion and mathematical induction. The latter two topics are heavily used in the specification and verification of regular circuits.

Digital design. You should be familiar with the behaviour of a reasonable selection of combinational and sequential gates and sub-systems. The circuits used in this text include: inverter, nand, nor, and, or, mux, xor, incrementer, adder, comparator, an alu, unit delay, flip-flops, register, shifter, and the counter.

Recommended texts: \cite{17, 36, 53, 57, 61, 72}.

Recursive definitions and proofs by induction. Several recursive data type definitions and proofs of functions over them were shown or set as exercises in part one on ML. HOL permits the definition of recursive data types and they are used to define implementations of regular hardware structures. For example, we may regard the base case, \texttt{adder 0} as a single full adder, and the induction case, \texttt{adder n+1}, as a full adder wired onto \texttt{adder n}, an adder of size \texttt{n}. Obviously we will use induction to verify
such regular sub-systems. In the case of the adder subsystem the base case verification is trivial. For the induction step, the primitive recursive definition of \texttt{adder n+1} is in terms of \texttt{adder n} wired to a full adder. The induction hypothesis allows us to assume that the implementation of \texttt{adder n} satisfies its specification. The full proof, which is not trivial, is given in chapter 10.

You should be familiar with one central theorem, the \textit{primitive recursion theorem}, which tells us that for any \( f \) with \( k+1 \) arguments \( n, x_1, x_2, \ldots, x_k \), (which we shorten to \( x \)), there are unique functions \( g \) and \( h \) such that

\[
\begin{align*}
0 x &= g x \\
n x &= h(n x, n, x)
\end{align*}
\]

\textbf{Table 4.2} Primitive recursion

As examples, in the case \( f = \text{add} \), \( g = \text{I} \) the identity function, and \( h = \text{SUC} \)

\[
\begin{align*}
\text{add} \; 0 \; x &= \text{I} \; x \\
\text{add} \; (\text{SUC} \; n) \; x &= \text{SUC} \; (\text{add} \; n \; x)
\end{align*}
\]

and in the case of \( f = \text{mult} \), \( g = \text{ZERO} \), and \( h = \text{ADD} \)

\[
\begin{align*}
\text{mult} \; 0 \; x &= \text{ZERO} \; x \\
\text{mult} \; (\text{SUC} \; n) \; x &= \text{add} \; (\text{mult} \; n \; x) \; n
\end{align*}
\]

HOL contains within itself a function \texttt{PRIM\_REC} which takes two arguments: a base definition and the inductive case. The form exhibited in table 4.2 is represented in HOL by

\[
f = \text{PRIM\_REC} (\lambda x . \text{g} x) (\lambda f n x . \text{h}(f n x, n, x))
\]

In this format,

\[
\begin{align*}
\text{add} &= \text{PRIM\_REC} (\lambda n . n) (\lambda f m n . \text{SUC} \; (f n)) \\
\text{mult} &= \text{PRIM\_REC} (\lambda n . 0) (\lambda f m n . \text{add} \; (f n) \; n)
\end{align*}
\]

Get used to the notation: you will find it echoed back by the HOL system when you ask to be reminded of function definitions.

\textbf{Recommended texts:} [12, 15, 33, 54].
In this section we are going to carry out some proofs with the HOL system. We designed $p\,HOL$ to look pretty much like HOL so there is not much to unlearn. The basic theorems and inference rules we used in $p\,HOL$ are theorems and inference rules in HOL. The major differences are that terms in HOL appear in double quotes and are parsed by the system. An immediate benefit is that we not use the double back-slash in either $\land$ or $\lor$. For example, $p\,HOL$'s `a $\land$ b $\Rightarrow$ (c $\lor$ d)` will be written "a $\land$ b $\Rightarrow$ (c $\lor$ d)" in HOL.

Other changes are systematic: the type $\text{TERM}$ of $p\,HOL$ becomes $\text{term}$ in HOL, and the type $\text{THM}$ of $p\,HOL$ becomes $\text{thm}$ in HOL. The $\text{mk}$ and $\text{dest}$ functions of $p\,HOL$ have their exact parallels in HOL. In HOL, they are written with $\text{mk}$ and $\text{dest}$ as prefixes, for example, $\text{mk\_thm}$, $\text{mk\_neg}$, $\text{mk\_imp}$, $\text{mk\_conj}$, ... and $\text{dest\_thm}$, $\text{dest\_neg}$, $\text{dest\_imp}$, $\text{dest\_conj}$, ...

To give you the flavour, here are some instructive proofs in HOL which also serve to introduce anti-quotation (or how to get at ML definitions from HOL) and the rules for quantifiers.

Example 4.5.1 A derived rule of inference.

Doing proofs from the axioms and inference rules is rather tedious. In order to take larger steps and shorten the proofs, we may not only build a library of theorems, but may also derive stronger inference rules. Here is a simple example.

\begin{align*}
\text{AND\_IMP\_RULE: thm} & \rightarrow \text{thm} \\
G \vdash (a \Rightarrow (b \Rightarrow c)) & \\
\text{--------------------------} \\
G \vdash (a \land b) \Rightarrow c
\end{align*}

We first show you how to go about constructing a proof (by experimentation) and then how to construct the real proof from the experiment.

We start by assuming the upper line of the inference rule, namely "a $\supset$ (b $\supset$ c)" and then introduce a new theorem by $\text{th2} = \text{ASSUME} "a \land b"$. Operating on $\text{th2}$ with $\text{CONJUNCT1}$ and $\text{CONJUNCT2}$ gives us two more theorems, $\text{th3} = a \land b \vdash a$ and $\text{th4} = a \land b \vdash b$. Now we can use $\text{MP}$ twice on $\text{th1}$ using each of these two theorems in turn to eliminate $a$ and $b$ on the right hand side and obtain $\text{th6} = a \land b$, a $\supset$ (b $\supset$ c) $\vdash c$. The last step is $\text{DISCH} "a \land b"$ $\text{th6}$.
What we are really after is a function that will take a theorem of the form \( a \supset (b \supset c) \) for any terms \( a, b, \) and \( c \) and turn it into a theorem of the form \( (a \land b) \supset c \) for the same terms. We have merely shown that this is true for variables \( a, b, \) and \( c, \) but it is easy to transform our working into a genuine inference rule. We write the rule as a function with one argument, a theorem, say \( \text{th1}. \) We split the argument into an assumption list and a conclusion by \text{dest_thm}. The conclusion is expected to take the form \( a \supset (b \supset c) \). If so we can decompose it into its constituent terms using \text{dest_imp} twice. Thus if the argument were

\[
\text{th1} = \vdash (x > 0) \supset (y > 0 \supset x + y > 0)
\]

then after executing

```plaintext
let (asm, concl) = dest_thm th1 in
let (a, rhs) = dest_imp concl in
let (b, c) = dest_imp rhs in
```

we would have that

- \( \text{asm} = [ ] \) (the assumption list is empty),
- \( \text{a} = "x > 0" \)
- \( \text{rhs} = "y > 0 \supset x + y > 0" \)
• \( b = \text{"} y > 0\text{"}, \) and

• \( c = \text{"} x + y > 0\text{"}. \)

Failure is reported if \( \text{th1} \) is not of the expected form.

Except for one detail, we can now code the definition of \texttt{imp_and_rule}. This detail concerns how we access ML variables from within a HOL term. If we write \texttt{ASSUME "a \& b"} we get back the theorem \( a \land b \vdash a \land b \). We actually want the ML terms referred to by \( a \) and \( b \). In this case, the theorem we require is \( (x > 0) \land (y > 0) \vdash (x > 0) \land (y > 0) \). The problem is solved by \texttt{anti-quotation}. When we refer to an ML variable from within a HOL term we prefix it with a "hat" — in this case we write \texttt{ASSUME "\( ^\hat{a} \land ^\hat{b} \)"}.

```ml
#let imp_and_rule th1
    = let (ass, concl) = dest_thm th1 in
      let (a, rhs) = dest_imp concl in
      let (b, c) = dest_imp rhs in
      let th2 = ASSUME "\( ^\hat{a} \land ^\hat{b} \)" in
      let th3 = CONJUNCT1 th2 in
      let th4 = CONJUNCT2 th2 in
      let th5 = MP th1 th3 in
      let th6 = MP th5 th4
      (DISCH "\( x > 0 \land y > 0 \) th6);;
    imp_and_rule = - : (thm -> thm)
```

We now give three illustrative cases of this rule in action: the first is our previous example, the second is an attempt on a theorem with the wrong structure which fails, and the third is a rather more complicated example. Notice that when HOL returns a theorem, only the conclusion is shown explicitly. The hypotheses are shown as dots (one for each individual hypothesis) to the left of the turnstile. The built-in function \texttt{hyp} can be used to access them. Similarly, \texttt{concl} returns the conclusion of a theorem as a term.
4.6 Rules for quantifiers

GEN: \[ \text{term} \rightarrow \text{thm} \rightarrow \text{thm} \]
\[
\frac{G \vdash \text{A}}{G \vdash \forall x. x = A}
\]

where \( x \) is not free in \( G \)

GENL: \[ \text{term list} \rightarrow \text{thm} \rightarrow \text{thm} \]
\[
\frac{[\ "x_1; x_2; \ldots; x_n\"]}{G \vdash \forall x_1 x_2 \ldots x_n . A}
\]

where \( x_1, x_2, \ldots, x_n \) are not free in \( G \)

GEN (short for generalise) takes a variable say \( x \) and a theorem say \( G \vdash A \) and returns \( G \vdash \forall x . A \) as a theorem. A well known example would be

\[
\text{GEN } "n:num" \ (\vdash P n \supset P (n+1)) = (\vdash \forall n . (P n \supset P (n+1)))
\]

which is used implicitly when carrying out proofs by mathematical induction. Notice the proviso that the variable argument must not be free in the hypotheses. The request GEN "x:bool" (\( x \vdash x \)) fails because \( x \) is free on the assumption list. HOL allows a greater range of variables than does \( p \text{ HOL} \), and insists that they be explicitly typed when a type cannot be
deduced from the immediate term containing it. In particular, an explicit
type must be supplied when the term is a single variable. **GENL** is used
when we have several generalisations to make.

```
#let th1 = ASSUME "x:bool";;
th1 = \x.

#let th2 = GEN "t:bool" th1;;
th2 = \t.

#GEN "x:bool" th1;;
evaluation failed

#let th3 = DISCH "x:bool" th1;;
th3 = \x. x ==> x

#let th4 = GEN "x:bool" th3;;
th4 = \x. x ==> x

#let th5 = ASSUME "p:bool";;
th5 = \p.

#let th6 = DISJ1 th5 "q:bool";;
th6 = \p \ q

#let th7 = DISCH "p:bool" th6;;
th7 = \p. p ==> p \ q

#let th8 = GEN "p:bool" (GEN "q:bool" th7);;
th8 = \p q. p ==> p \ q

#let th9 = GENL [ "p:bool"; "q:bool" ] th7;;
th9 = \p q. p ==> p \ q
```

**SPEC**: term -> thm -> thm
```
   "t"
   G |- x : A

   G |- A [ t/x ]
```

where `t` is free for `x` in `A`.

**SPECL**: term list -> thm -> thm
```
   [ "t1"; "t2"; ... "tn" ]
   G |- ! x1 x2 ... xn : A

   G |- A [ t1/x1, t2/x2, ..., tn/xn ]
```

where the `t`'s are free for the `x`'s in `A`.

**SPEC** (short for specialise) works in the opposite direction. Given a
quantified theorem `th1 = G |- \ x . body`, the request **SPEC** "term" `th1`
CHAPTER 4. THE HOL NOTATION

checks that the types of \texttt{term} and \texttt{x} are the same, and then returns \( G \vdash \text{body \ [ term / x ] } \) as a theorem (each free occurrence of \texttt{x} in the body will be replaced by \texttt{term} with systematic name conversion to prevent name capture).

\begin{verbatim}
#SPEC "x:bool" (ASSUME "! x . x");
   \texttt{. \ LO} x

#SPEC "y:num" (ASSUME "! y . x <= x + y");
   \texttt{. \ LO} y, y <= (y + y)
\end{verbatim}

\texttt{SPECL} is an obvious extension that allows several substitutions to be carried out at the same time.

\begin{verbatim}
#let th10 = SPEC "a:bool" th9;
\texttt{th10 = \ LO} \texttt{q} . a ==> a \lor q

#let th11 = SPEC "(a \lor c)" th10;
\texttt{th11 = \ LO} a ==> a \lor a \lor c

#let th12 = SPECL \[ "a:bool" /; "a \lor c" /; \] th9;
\texttt{th12 = \ LO} a ==> a \lor a \lor c
\end{verbatim}

We need \texttt{EXISTS} and \texttt{CHOOSE} to convert between a theorem like \( \vdash \exists p . (p = \sim x) \land (y = \sim p) \) and a theorem \( \vdash (\sim x = \sim x) \land (y = \sim x) \) (which is easy to simplify further).

\begin{verbatim}
EXISTS: (term \# term) \to \text{thm} \to \text{thm} 
\[ "? x . A \ [ x ]", "t"
\] 
\[ G \vdash A \ [ t ] \]
\[ \] 
\[ G \vdash ? x . A \ [ x ] \]
\end{verbatim}

where the substitution of \texttt{x} for some occurrences of \texttt{t} in \texttt{A} does not cause any \texttt{x} to become bound

\begin{verbatim}
CHOOSE: (term \# thm) \to \text{thm} \to \text{thm} 
\[ (t, G1 \vdash ? x . A \ [ x ] \]
\[ G2 + "A \ [ v ]" \vdash A' \]
\[ \] 
\[ G1+G2 \vdash A' \]
\end{verbatim}

where the variable \texttt{v} occurs nowhere in \texttt{G2} or in \texttt{A'}

\texttt{EXISTS} is used to introduce an existentially quantified variable, e.g., we can use \texttt{EXISTS} to prove \( \vdash \ ? n . \ P n \) from \( \vdash P \ 1 \). The introduced quantified variable will replace zero or more occurrences of a specific term in a given
4.6. RULES FOR QUANTIFIERS

theorem, i.e. from \( \vdash \Pi \, P \, \equiv A \), we may prove any of \( \vdash ? \, n \, . \, P \, n \, \equiv A \) and \( \vdash ? \, n \, . \, P \, \equiv A \). Thus we have to be able not only to nominate the quantified variable we are introducing, but also state precisely where we want it to appear in the final result.

EXISTS : \(((\text{term} \# \text{term}) \rightarrow \text{thm} \rightarrow \text{thm})\) takes a pair of terms \((t_1, t_2)\), a theorem \( \vdash t_1, t_1 \) gives the form of the final result (we return \( \vdash t_1 \)). \( t_2 \) is term in \( t_1 \) we are replacing by the quantified variable, and \( t_1 \) is the theorem we are transforming. Given that \( t_1 \) has the form \( \forall \alpha . \, \text{body} \), if a call EXISTS \((t_1, t_2) \) \( (\vdash t_1) \) can show that \( \text{body}[t_2/\alpha] = t_1 \), then \( \vdash t_1 \) is returned. If not, the call fails. Here are some examples:

```
#let th1 = REFL "(P:num->bool) 1";;
  th1 = \vdash P \, 1 \, \equiv P

#EXISTS("? n . (P:num->bool) n \, 1 \, \equiv P \, n", "!") th1;;
  \vdash ?n. P \, n \, \equiv P

#EXISTS("? n . (P:num->bool) n \, 1 \, \equiv P \, n", "!") th1;;
  \vdash ?n. P \, n \, \equiv P

#EXISTS("? n . P \, 1 \, (P:num->bool) n \, 1", "!") th1;;
  \vdash ?n. P \, n \, \equiv P
```

CHOOSE is used to remove an existentially quantified variable by substitution. Given a theorem like \( \vdash ? \, x . \, y \, = \, y \, \land x \), we can use CHOOSE to instantiate a the specific example, say \( x = T \), and return \( \vdash y = y \). We must take care to guard against “wrong” choices, e.g. \( x = F \), which is why the form of CHOICE is rather detailed.

CHOOSE : \(((\text{term} \# \text{thm}) \rightarrow \text{thm} \rightarrow \text{thm})\) takes a (term \# theorem) pair \((t_1, \text{th1})\) and a theorem \text{th2} as arguments. \( t_1 \) is the substitution, \( \text{th1} = \vdash ? \, x . \, A \, [x] \) is the theorem in which we are going to effect the substitution and \( \text{th2} = A \, [v] \) \( \vdash B \) is the result we are after. What CHOOSE has
to do is show that $A \ [ \forall x \ ] \supset B$. If so then $\vdash B$ is returned. Here is an example

```haskell
let th3 = assume "\? p. (p = 'x) /\ (y = '~p)" ;;
th3 = . |- ?p. (p = 'x) /\ (y = '~p)

let th4 = assume "(z = 'x) /\ (y = '~z)" ;;
th4 = . |- (z = 'x) /\ (y = '~z)

(conjunct1 th4, conjunct2 th4);
(. |- z = 'x, . |- y = '~z) : (thm # thm)

(hyp (conjunct1 th4), hyp (conjunct2 th4));
(['(z = 'x) /\ (y = '~z)'], ['(z = 'x) /\ (y = '~z)'])
: (term list # term list)

let th5 = rewrite_rule [conjunct1 th4] (conjunct2 th4);

th5 = . |- y = x

hyp th5;;
['(z = 'x) /\ (y = '~z)'] : term list
```

where `rewrite_rule [thms] th` rewrites `th` with a list of equality theorems (here just $\vdash z = 'x$) and carries out some simplifications (here simplifying $y = '~(x)$ to $y = x$).

```haskell
choose ("z:bool", th3) th5;;
 . |- y = x

hyp it;;
['?p. (p = 'x) /\ (y = '~p)" : term list
```

Both these rules are needed in example 4.6.3.

**Example 4.6.2** Prove $\vdash (\forall (x:bool) . A x) \supset (\exists x . A x)$

The proof of this theorem is quite simple but nonetheless gives us examples of `spec` and `exists` in use. We start by `assume "! x . A x"`. We eliminate the quantifier by `spec` to produce `th2 = ! x . A x ⊢ A x` and then apply `exists ("? x:bool . A x", "x:bool")` to `th2`. All that now remains is to discharge the assumption.
4.6. RULES FOR QUANTIFIERS

Example 4.6.3 EXISTS_ELIM

Prove that $\forall t \exists x . (x = t \land \forall x)$, where $t$ is free for $x$ in $\forall x$.

This is easily our toughest theorem so far. It is worthwhile for two reasons: (i) variants of it are used extensively in hardware verification, and (ii) it is reasonably large and shows up the shortcomings of the forward proof style where the prover has to keep track of the state of the proof and ensure that all cases have been covered. These drawbacks give us a nice motivation for HOL style “backwards” proofs.

As is typical for proofs of equivalence, we show implication in both directions — i.e., that $\vdash \forall t \supset (\exists x . (x = t) \land \forall x)$ and $\vdash (\exists x . (x = t) \land \forall t) \supset \forall t$ — and then use `IMP_ANTISYM_RULE` to infer the desired result.

$\vdash \forall t \supset (\exists x : \text{bool} . (x = t) \land \forall x)$

We start by getting $\forall t$ onto the assumption list (for later discharging) by `ASSUME "t"`. Now we build up the right hand side into a suitable form.
First we introduce a new theorem $\vdash t = t$, make its assumptions the same as those of the first theorem using \texttt{ADD\_ASSUM "A t" (REFL "t")}, and then use \texttt{CONJ} to produce $\texttt{th3} = \vdash (t = t) \land A \ t$. By applying \texttt{EXISTS ("? x:bool . (x = t) \land A x", "t:bool")} to \texttt{th3} we manufacture exactly the right hand side we have been looking for, namely $\exists x . (x = t) \land A x$ (the value of $x$ in question is of course $t$) and a simple discharge produces the theorem we are after $\vdash A t \supset \exists x . (x = t) \land A x$.

$\vdash (\exists (x:bool) . (x = t) \land A x) \supset A t$

Just as we used \texttt{EXISTS} in the first part of the proof, you might guess that we are going to use \texttt{CHOOSE} in the second part. We manufacture two auxiliary theorems
\[
\texttt{th13 = } \exists x . (x = t) \land A x \vdash A t \text{ (which is hard)}\text{ and th10 = } \exists x . (x = t) \land A x \vdash \exists x . (x = t) \land A x \text{ (trivial) from which, making use of CHOOSE, we may infer that } \exists x . (x = t) \land A x \vdash A t .
\]

Discharging this theorem and combining it with \texttt{th5} via \texttt{IMP\_ANTISYM\_RULE} leads to the desired result.

```isabelle
#let th6 = ASSUME "? x:bool . (x=t) \land A x";;
th6 = . |- ?x. (x = t) \land A x

#let th7 = ASSUME "(b=t) \land (A:bool->bool) b";;
th7 = . |- (b = t) \land A b

#let th8 = CONJUNCT1 th7;;
th8 = . |- b = t

#let th9 = CONJUNCT2 th7;;
th9 = . |- A b

#let th10 = AP\_TERM "A:bool->bool" th8;;
th10 = . |- A b = A t

#let th11 = fst (EQ\_IMP\_RULE th10);;
th11 = . |- A b ==> A t

#let th12 = MP th11 th9;;
th12 = . |- A t

#let th13 = CHOOSE ("b:bool", th6) th12;;
th13 = . |- A t

#let th14 = DISCH "?x. (x=t)/(A:bool->bool)x" th13;;
th14 = . |- (?x. (x = t) \land A x) ==> A t

#let th15 = IMP\_ANTISYM\_RULE th5 th14;;
th15 = . |- A t = (?x. (x = t) \land A x)
```
We start by assuming \( T_1 \) and \( T_2 \) as theorems \( \text{th6} \) and \( \text{th7} \) respectively. Applying \( \text{CONJUNCT1} \) to \( \text{th7} \) results in \( \text{th8} \models T_2 \models (b = t) \). If \( b = t \) then \( \mathbb{A} b = \mathbb{A} t \) by \( \text{AP_TERM} \) and if we have equality, then we also have implication both ways. Hence using \( \text{EQ_IMP_RULE} \) we get \( \text{th11} = T_2 \models \mathbb{A} b \supset \mathbb{A} t \). Applying \( \text{MP} \) to \( \text{th12} \) and the right conjunct of \( \text{th7} \) returns \( \text{th12} = T_2 \models \mathbb{A} t \). The rest of the proof is obvious. Using \( \text{CHOOSE} \) on \( \text{th12} \) and then \( \text{DISCH} \) \( T_2 \) on that returns \( \text{th14} = T_2 \supset \mathbb{A} t \). The final step in the proof is to combine theorems \( \text{th5} \) and \( \text{th14} \) by \( \text{IMP_ANTISYM_RULE} \) to finish the proof. In the script version of the proof, we let \( T_1 = \exists x . (x = t) \land \mathbb{A} x \) and \( T_2 = (b = t \land \mathbb{A} b) \) for short.

### 4.7 Backward proof

As exemplified by the last proof, the forward method of proof suffers from two handicaps: (i) it is usually quite hard to know where to start, and (ii) in general, there is a lot of bookkeeping associated with large proofs. HOL supports forward proof, but also has facilities for backward proof. In a backward proof, we start by supplying what we want to prove as a goal. Goals may be rewritten or split into simpler parts by tactics. Tactics may be thought of as inverse inference rules. A tactic splits a goal into subgoals but we are assured that if we can prove each of the subgoals then there is a companion inference rule to that tactic which will prove the initial goal from these proofs. Suppose tactic \( T \) takes goal \( G \) to subgoals \( g_1 \) and \( g_2 \). Then there will be an inference rule \( R \) which takes theorems \( \vdash g_1 \) and \( \vdash g_2 \) to the theorem \( \vdash G \). For example, there is an inference rule \( \text{IMP_ANTISYM_RULE} \) which constructs the theorem \( \vdash a = b \) from the theorems \( \vdash a \supset b \) and \( \vdash b \supset a \). Corresponding to \( \text{IMP_ANTISYM_RULE} \) there is a tactic \( \text{EQ_TAC} \) which when applied to the goal \( a = b \) produces two subgoals \( a \supset b \) and \( b \supset a \). The idea behind a backward proof in HOL is that we recursively apply tactics to decompose a goal into a tree of subgoals until all the leaves are simple enough to prove. Then the HOL system will take our manually generated pattern of tactics and subgoals and transform it into a forward proof. HOL supplies several functions to manipulate goals. These include:

- \( \text{g:term} \rightarrow \text{goal} \) is used to initialise a goal stack to the next theorem to be proved. The older variant \( \text{set_goal:assums#term} \) may also be used, but usually the assumption list is empty so we might as well use \( g \).

- \( \text{e = expand} \) takes a tactic and applies it to the current goal;
- \textbf{b = backup} backs up the stack (throws away the last application of a tactic) and is very useful during proof explorations.

- \textbf{r n = rotate n} which rotates the order of the sub-goals by n.

If a tactic splits a goal into more than one goal (as with \texttt{EQ\_TAC}) the subgoals are tackled one at a time. When one subgoal has been proved, the next to be proved is popped from the top of the goal stack.

\textbf{Example 4.7.4} \vdash (\exists (x:bool). (x = t) \land A x) = A t

As an example, here is the proof in HOL of a primitive version of a theorem which is used to remove hidden wires in hardware proofs. We start by setting the goal and then split it into two implications using \texttt{EQ\_TAC}.

```
# g "(\forall x:bool . (x = t) \land A x) = A t";;
"(\forall x. (x = t) \land A x) = A t"

() : void
```

```
# e(EQ\_TAC) ;;
OK ...
2 subgoals
"A t =\rightarrow (\forall x. (x = t) \land A x)"
"(\forall x. (x = t) \land A x) = A t"

() : void
```

It is easier to work with the implicands on the assumption list. The tactic \texttt{STRIP\_TAC} takes the antecedent of an implication and pushes it onto the assumption list: Since we would like this to happen for both subgoals, we back up and apply the compound tactic \texttt{EQ\_TAC THEN STRIP\_TAC} to the original goal.

```
# b();;;
"(\forall x. (x = t) \land A x) = A t"

() : void
```

```
# e(EQ\_TAC THEN STRIP\_TAC) ;;
OK ...
2 subgoals
"(\forall x . (x = t) \land A x"
  [ "A t"
  [ "x = t"
  [ "A x"

() : void
```
The pertinent assumptions are listed under each subgoal. We work on the subgoals one at a time. The textually lower one takes precedence.

As can be seen, STRIP_TAC has “chosen” (pun) to simplify the existentially quantified antecedent. We can use both the assumptions to attack the goal. A simple-minded method is to discharge $\forall x$ and then rewrite what we have with the assumption $x = t$.

```plaintext
#e (UNDISCH_TAC "(A:bool->bool) x");
OK...
"\forall x \implies A t"
 [ "x = t" ]

() : void
```

```plaintext
#e (ASM_REWRITE_TAC []);
OK...
goal proved
. |- \forall x \implies A t
.. |- A t

Previous subproof:
"?x. (x = t) /\ \forall x"
 [ "A t" ]

() : void
```

ASM_REWRITE_TAC [] uses terms in the assumption list to perform rewrites, together with a few rules that are applied automatically. Extra theorems may be included explicitly in its argument list. The other subgoal is now popped. In this case we choose a suitable value for $x$ (if we choose a silly value, the goal will be rendered unprovable, but the system remains sound) and the proof is completed with a simple rewrite from the assumption list.

```plaintext
#e (EXISTS_TAC "t:bool");
OK...
"(t = t) /\ A t"
 [ "A t" ]

() : void
```
The session is completed by storing away the theorem in the current theory under the name \texttt{EXISTS\_ELIM}. \texttt{prove\_thm} is a built-in theorem which takes a save name (in single back quotes), a goal (in double quotes), and a tactic from which it generates a forward proof of the theorem.

Note that when a tactic splits a goal into several subgoals we supply suitable tactic for each of the subgoals in a list following a \texttt{THENL}. When each subgoal tactic has a common tail (here \texttt{ASM\_REWRITE\_TAC \[]}) we can merge these subgoal branches.
Chapter 5

Verifying hardware

Over the last dozen or so years, rapid advances in technology have brought hardware devices into the everyday life of the general public. We find chips used in systems as diverse as: watches and pocket calculators; bank card systems; controllers in cars and domestic appliances; railway signalling and flight controllers. These devices go wrong from time to time. Sometimes this can be shrugged off, sometimes it cannot. For example, a watch that gains or loses a little time can always be reset; and if a pocket calculator runs amok, we can always switch it off and start the computation again. Neither would cause us too much concern, except that we might reflect on our chances of building a large system correctly when such simple ones go wrong. Bank card systems occasionally hang up, or swallow a card and keep it. This is annoying and inconvenient, especially if the incident happens out of banking hours. More serious problems are occasioned by faults in chips that control certain functions in cars, or in domestic appliances, such as washing machines. The cost would be very high if the fault were major and that model of car or appliance had to be recalled. The fourth category — safety critical chips — is quite a different matter. These systems are usually substantial mixed systems of software and hardware and the repercussions can be very serious if the controller is flawed — bystanders, passengers and crew may be injured or die.

We would thus like to be able to guarantee that a design does what it is supposed to, neither more nor less. This we do in three major steps:

- **specify the intent of the design** clearly and unambiguously by giving relations which relate the outgoing signals values to current and previous input signals. A good specification is concerned only with the net effect of a design and not how it is constructed. Thus specifications should be entirely mathematical in nature.

- **define the implementation** clearly and unambiguously, typically by listing its (simpler) constituents and indicating how they are wired together.

- **verify that the implementation does indeed meet the specification** (to within the tolerance of some underlying model).
If we succeed we will then have a correctness statement (a theorem) which states something like

$$\forall n \; \text{inputs outputs} . \; \text{implementation } n \equiv \text{specification } n$$

i.e. over all generic sizes, inputs and outputs, the implementation meets the specification exactly. Sometimes designs are subject to certain constraints. A typical constraint might be that two inputs are related, say one is the always inverse of the other. In which case the theorem to be proved will have the form

$$\forall n \; \text{inputs outputs} .$$
\[ \text{constraints } \supset ( \text{implementation } n \equiv \text{specification } n) \]

which states that the implementation meets the specification provided that the constraints hold. Constraints help to simplify a proof obligation by removing certain cases from consideration.

Weaker forms of correctness use logical implication:

$$\forall n \; \text{inputs outputs} .$$
\[ \text{implementation } n \supset \text{specification } n \]

$$\forall n \; \text{inputs outputs} .$$
\[ \text{constraints } \supset ( \text{implementation } n \supset \text{specification } n) \]

Relaxing the correctness criterion from equivalence to implication allows the specification to stipulate only selected properties of an implementation's behaviour. Tom Melham's PhD thesis [80] is very strong on the formulation of correctness statements and abstractions. Read it!

The devices we prove now will be used later as building blocks in the construction of larger systems. Their correctness statements enable us to replace the detail and volume of the implementation by the distilled and mathematical specification wherever a verified device is used as a building block. Once a piece of hardware has been specified and verified, the specification, implementation and correctness proof can be saved in a verification library. When that hardware is used later in a more complex design, we don't need to know how it was constructed, we can just work with its specification in the new verification. It pays therefore to give some thought to shaping a specification into a form that is convenient for later manipulation.
even if this may make the immediate verification a little harder. After all, a verification is carried out but once, but the specification may be needed many times. It follows that when working in a specific area, say arithmetic sub-systems, we should try to find a template for the specifications so that they are of a similar nature. Then techniques used in one verification can often be used in the next.

In this chapter we show you how to specify hardware designs, define their implementations, and how to carry out paper and pencil verifications in the style of HOL. Before we set out to construct a device, we give it a specification. A specification treats a device as a black box and tells you what can be observed on each output line in terms of current and previous inputs. In other words, a specification tells you what a device does, but not how it does it — that is the job of an implementation definition. Since coining specifications is hard, we spend much of this chapter specifying a selection of straightforward combinational and sequential circuits and sub-systems. We have striven to make these specifications readable and generic. For example the specifications of the arithmetic gates and arithmetic sub-systems have the same structure even though the gate specifications use bit values on individual wires and arithmetic sub-systems are abstracted in terms of numerical values on busses. Using an appropriate yet general notation is important because it enables theorems and proof styles to be used across a number of verifications.

Having covered a number of specifications, we next show how implementations are defined. This turns out to be relatively easy: all we have to do is enumerate the constituent parts and how they are wired together. A key item in the definition of implementations is that when we do proofs we must be able to eliminate all occurrences of hidden (purely internal) lines from them. It would be nice if we could do this with an existing rule of formal logic and not have to invent something special for hardware, and this indeed is the case. All we need do is existentially quantify hidden lines in definitions of implementations and then we can use a variant of the rule \texttt{EXISTS\_ELIM}, proved in the last chapter, to accomplish our aim.

We close the chapter with some paper and pencil verifications of some relatively easy designs. These informal proofs (often called “proof outlines”) should give you a reasonable feeling for the HOL approach when we move into the realms of machine proof in succeeding chapters. When we do proofs we seek to show that an implementation meets its specification over all input and output values. Sequential proofs are harder than combinational proofs in that we must match signals on lines at all times of interest. Regular subsystems are given primitive recursive definitions and proofs of regular subsystems, whether combinational or sequential, succumb to mathematical
induction. Thus whatever the size of a subsystem, its correctness can be verified in at most two steps (a base case and an induction step).

5.1 Combinational circuits

5.1.1 Specification

Let the typical device $\text{dev}$ have $m$ input lines, say $i_1, i_2, ..., i_m$ and $n$ output lines, say $z_1, z_2, ..., z_n$ as shown in figure 5.1. (In the case of a chip, these $m + n$ external lines would correspond to its pins.) When the device is operational each line carries a value drawn from some set of possible values. In our case, since we will always be modelling strong signals, this will always be a bool value, either $T$ (high) or $F$ (low). Usually, each input value may be freely chosen from the set $\{T, F\}$, but the output values will be constrained. For example, in the case of an inverter, the output value will always be the inverse of the input value. Different hardware devices relate their outputs to their inputs in different ways — what we need is a consistent way of expressing these relations. We specify the behaviour of a device by defining a (usually curried) predicate $\text{DEV}$ with $m + n$ arguments

\[ \text{DEV} \ i_1 \ i_2 \ ... \ i_m \ z_1 \ z_2 \ ... \ z_n \ \equiv \ \text{defining relations} \]

which is true if and only if $i_1, i_2, ..., i_m, z_1, z_2, ..., z_n$ are allowable values on the corresponding lines of the device $\text{dev}$. Given a set of input values on $i_1, i_2, ..., i_m$, the predicate $\text{DEV}$ typically gives a relation for each output line $z_1, z_2, ..., z_n$ in terms of $i_1, i_2, ..., i_m$. $\text{DEV}$ will be true only in the cases when the relations on the right hand side correctly model the behaviour of the hardware box it represents. In the case of an inverter, the predicate will be $\text{inv} \ i \ z = (z = \sim i)$ (where, since $m = 1$ and $n = 1$, we have dropped the subscripts on the lines). Of the four possible combinations for the pair $(i, z)$, namely $(F, F), (F, T), (T, F), (T, T)$, the predicate $\text{inv} \ i \ z$ is true only for the pairs $(F, T)$ and $(T, F)$.
5.1. COMBINATIONAL CIRCUITS

Example 5.1.1 mux2 — the two input multiplexor

Figure 5.2 The mux2 gate

The truth table for the mux2 gate is shown in table 1. The gate has three inputs and one output, each carrying bool values.

<table>
<thead>
<tr>
<th>inputs</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>sel</td>
<td>a</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

Table 5.1 Multiplexor truth table

This is a rather long-winded way of stating that $z$ tracks $a$ if $sel$ is low and tracks $b$ if $sel$ is high. Here is a formal specification corresponding to this table:

$$\text{mux2_spec sel a b z = (z = sel => a | b)}$$

On the left hand side of the specification we uniquely identify the hardware device being specified $\text{mux2_spec}$ and list each input port and each output port ($sel$ $a$ $b$ $z$). By convention, we use the name of the port to stand for the signal value on that line. On the right, we state the defining relation for the device. In this case the signal on the output line is constrained to be in the relation $z = sel \Rightarrow a \mid b$ with its three inputs. We may deduce which ports are output ports and which are input ports by whether or not they appear on the left side of such relations. Much of the time we can infer the types of the ports from the body. Here we may infer that $sel$ must be
boolean, and $z$, $a$ and $b$ must be of the same type. In cases like this, HOL (unlike ML) expects us to type the ports explicitly. Although we may opt for these lines being polymorphic, it is sufficient for our current purposes that they be of type $\text{bool}$. We adopt the convention of doing the typing on the left hand side, as in

$$\text{mux2-spec sel (a:bool) (b:bool) (z:bool) = ...}$$

rather than

$$\text{mux2-spec sel a b z = ((z:bool) = sel => (a:bool) | (b:bool))}$$

Of course, we need not type more than one of $z$, $a$, or $b$ (choose any one) and HOL will infer the type of the other two variables, e.g.

$$\text{mux2-spec sel (a:bool) b z = ...}$$

**Example 5.1.2 The 1-bit full adder**

We now specify another much-used gate, the 1-bit full adder. By introducing an auxiliary function $bv$ we create a style of specification that generalises to other arithmetic circuits and will generalise further to arithmetic sub-systems.

![Full adder diagram](image)

The full adder has three input lines (here $a$, $b$ and $\text{cin}$) and two output lines (here $s$ and $c$). It is usual to define the behaviour of this device by a table which enumerates all the possibilities. This we have done in two styles in table 5.2. The first formulation sticks to the boolean viewpoint for expressing the values on the output lines. The second formulation looks at the value on the sum line $s$ “through sum eyes”. In order to do this we introduce a very simple piece of notation and define a transfer function $bv$ with the properties that $bv \, T = 1$ and $bv \, F = 0$. 
\(\text{bv b} = (b => 1 \mid 0)\)

\(\text{bv}\) is typed as \(\text{bool} \rightarrow \text{num}\). The sum of the input bits is now expressible as \(\text{bv a} + \text{bv b} + \text{bv cin}\). We may also express the value on the \(s\) line as a \(\text{num}\) (0 or 1) rather than as a \(\text{bool}\) (F or T).

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
<th>2nd formulation for the outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>a b cin</td>
<td>s c</td>
<td>sum mod 2</td>
</tr>
<tr>
<td>F F F</td>
<td>F F</td>
<td>0</td>
</tr>
<tr>
<td>F F T</td>
<td>T F</td>
<td>1</td>
</tr>
<tr>
<td>F T F</td>
<td>T F</td>
<td>1</td>
</tr>
<tr>
<td>F T T</td>
<td>F T</td>
<td>2</td>
</tr>
<tr>
<td>T F F</td>
<td>T F</td>
<td>1</td>
</tr>
<tr>
<td>T F T</td>
<td>F T</td>
<td>2</td>
</tr>
<tr>
<td>T T F</td>
<td>F T</td>
<td>2</td>
</tr>
<tr>
<td>T T T</td>
<td>T T</td>
<td>3</td>
</tr>
</tbody>
</table>

where \(\text{sum} = \text{bv a} + \text{bv b} + \text{bv cin}\)

Table 5.2 1-bit full adder truth table

Since we will be spending a lot of time working with specifications, it pays to find styles that generalise and can be used repeatedly. Not least because groundwork done on one example will often transfer to the next. What we are looking for here is a style that extends in a straightforward fashion to the 1-bit half adder, the 1-bit incrementer, and even the \(n\)-bit incrementer and \(n\)-bit adder and other arithmetic sub-systems. Table 5.2 gives us a hint on how to do this if we notice that the sum of the inputs, \(\text{sum} = \text{bv a} + \text{bv b} + \text{bv cin}\), decides the outputs.

- \(c\) is high only when the input bits \((a, b, \text{cin})\) sum to 2 or 3.
- \(s\) is low when the input bits \((a, b, \text{cin})\) sum to 0 or 2, be they \((0, 0, 0), (1, 1, 0), (1, 0, 1)\) or \((0, 1, 1)\). \(s\) is high when the input bits sum to either 1 or 3.

We can thus express our specification of the 1-bit full adder in terms of \(\text{sum}\). The simpler relation is that for \(c\), namely \(c = (\text{sum} \geq 2)\) (or \(c = (\text{sum div 2} = 1)\)). We cast the relation for \(s\) in terms of \(\text{numas}\) as \(\text{bv s} = (\text{sum mod 2})\). Accordingly we express our specification of the 1-bit full adder as:

\[
\begin{align*}
\text{fulladder a b cin s c} & \\
& = \text{let sum = bv a + bv b + bv cin} \\
& \text{in} \quad \text{bv s} = (\text{sum mod 2}) \land \\
& \quad \text{c} = (\text{sum} \geq 2)
\end{align*}
\]
The body of the specification supplies a relation for each output line. Only when both these relations are satisfied at the same time will we have a full adder, which is why we conjoin them on the right hand side. In words, the specification tells us that the value on the sum line \( s \) is the sum of the values on the three input lines \( \mod 2 \); and the value on the carry line \( c \) is high if at least two of the three input lines are high, otherwise it is low.

Here are the specifications of some other common arithmetic circuits in the same style:

\[
\text{half_adder } a \ b \ s \ c \\
= \begin{array}{l}
\text{let } \text{sum} = \text{bv} \ a + \text{bv} \ b \\
\text{in } \text{bv} \ s = (\text{sum mod } 2) \land \\
\text{c} = (\text{sum } \geq 2)
\end{array}
\]

\[
\text{twoComp } a \ b \ s \ c \\
= \begin{array}{l}
\text{let } \text{sum} = \text{bv} \ a + \text{bv} \ b \\
\text{in } \text{bv} \ s = ((2 - \text{sum}) \mod 2) \land \\
\text{c} = (\text{sum } \neq \text{0})
\end{array}
\]

**Example 5.1.3 nAdder — an adder sub-system**

Not unexpectedly, specifying gates is straightforward. We now move onto specifying arithmetic sub-systems using the well-known ripple carry adder as example. Once again we seek a style that will generalize to other arithmetic sub-systems. The key is to find a way of lumping bus wires together and inventing a notation that enables us to manipulate the value on the bus as a whole rather than by bit values on each of its individual constituent wires.

Our word adder sub-system has three input ports \( a, b, \) and \( \text{cin} \), and two output ports \( s \) and \( c \). (By convention, we will write buses in bold face in this chapter.)
5.1. COMBINATIONAL CIRCUITS

Figure 5.5  Buss with n wires

The lines of the input bus $a$ are individually numbered $a_0, a_1, ..., a_n$; of the input bus $b$ are individually numbered $b_0, b_1, ..., b_n$; and the lines of the output bus $s$ are individually numbered $s_0, s_1, ..., s_n$.

We call this device $n$-Adder $n a b cin s c$ — remember that the $n$-Adder has $n+1$ bit lines per bus since we start numbering at zero. We say that $n$-Adder $n a b cin s c$ has size $n+1$. It expects a $num$ input on bus $a$, a $num$ input on bus $b$, a $bool$ value on the carry-in line $cin$, and outputs a $num$ on bus $s$ and a $bool$ on the carry-out line $c$. The bus lines $a, b$ and $s$ carry signals which we will interpret as numbers in the range 0 through $2^{n+1} - 1$ inclusive.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
<th>2nd formulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_i$</td>
<td>$a_0$</td>
<td>$b_1$</td>
</tr>
<tr>
<td>T T F F F</td>
<td>T T F</td>
<td>T T T T T T</td>
</tr>
<tr>
<td>T T F F F</td>
<td>T T F</td>
<td>T T T T T T</td>
</tr>
<tr>
<td>T T F T T</td>
<td>T T F</td>
<td>T T T T T T</td>
</tr>
<tr>
<td>T T F F T</td>
<td>T T F</td>
<td>T T T T T T</td>
</tr>
<tr>
<td>T T T T T</td>
<td>T T F</td>
<td>T T T T T T</td>
</tr>
<tr>
<td>T T T F T</td>
<td>T T T</td>
<td>T T T T T T</td>
</tr>
<tr>
<td>T T T F T</td>
<td>T T T</td>
<td>T T T T T T</td>
</tr>
<tr>
<td>T T T F T</td>
<td>T T T</td>
<td>T T T T T T</td>
</tr>
<tr>
<td>T T T T T</td>
<td>T T T</td>
<td>T T T T T T</td>
</tr>
<tr>
<td>T T T F T</td>
<td>T T T</td>
<td>T T T T T T</td>
</tr>
<tr>
<td>T T T T T</td>
<td>T T T</td>
<td>T T T T T T</td>
</tr>
<tr>
<td>T T T T T</td>
<td>T T T</td>
<td>T T T T T T</td>
</tr>
</tbody>
</table>

where $SUM = 2^0 a_0 + 2^1 (b_0 a_1) + 2^2 b_0 + 2 x (b_0 b_1) + b_0 cin$

Table 5.3  2-bit adder truth table
If we follow our specification style for circuits, we might be tempted to seek a specification for this sub-system as a set of equations,

\[(s_0 = ... ) \land (s_1 = ... ) \land (s_n = ... ) \land (c = ... )\]

one for each of the \(s_k\)’s individually and for the carry line \(c\). There must be a better way! Table 5.3 tabulates behaviour of a 2-bit adder subsystem (the first 8 rows and the last 8 rows only) and points us towards a suitable abstraction. Columns 1–5 show the input bits, columns 6–8 display the output bits separately, column 9 their SUM, i.e.

\[SUM = bv \ a_0 + 2 \times (bv \ a_1) + bv \ b_0 + 2 \times (bv \ b_1) + bv \ cin,\]

and columns 10 and 11 show our adopted abstraction. We group associated wires together into a bus and work with the \(num\) they represent. Different combinations of values on the \(a\), \(b\) and \(cin\) lines produce different output effects, but the outputs will be the same for the same \(SUM\) value (column 9) however it arises.

By noting that \(SUM = (bv \ s_0) + 2 \times (bv \ s_1) + 4 \times (bv \ c)\), it is easy to specify the outputs of a 2-bit adder sub-system in terms of \(SUM\).

\[
\begin{align*}
    bv \ s_0 + 2 \times (bv \ s_1) &= (SUM \ mod \ 4) \land \\
    c &= (SUM \leq 4)
\end{align*}
\]

We define an abstraction which generalises this approach. \(val \ f \ n\) takes in the \(n + 1\) bit values on individual wires in a bus \(f\) (representing them by \((f \ 0), (f \ 1), \ldots, (f \ n)\)) and interprets them as the \(num\)

\[bv(f \ 0) + 2 \times bv(f \ 1) + 4 \times bv(f \ 2) \ldots + 2^n \times bv(f \ n)\]

Thus, for example, \(bv \ s_0 + 2 \times (bv \ s_1) + 4 \times (bv \ s_2)\) can be written \(val \ s \ 2\). \(val\) is defined by primitive recursion:

\[
\begin{align*}
    val \ f \ 0 &= f \ 0 \\
    val \ f(n + 1) &= val \ f \ n + 2^{n+1} \times (bv(f(n + 1)))
\end{align*}
\]

The mapping may be inverted. That is, given a number, we may retrieve the individual bits uniquely, so no information is lost by the adoption of this abstraction. When the vector \(f\) has size 0, its \(num\) interpretation is the value represented by the bit value of its one bit \(f_0\). When the vector \(f\) has size \(n + 1\) its bits are numbered from 0 through \(n + 1\), and its value is the value of its least significant \(n\) bits \((val \ f \ n)\) plus \(2^{n+1}\) times the bit value of the most significant bit (numbered as the \(n + 1^{th}\)). As an example, if \(f_0 = F, f_1 = T, \) and \(f_2 = T\), then
The formal specification of the nAdder reads:

\[
\text{nAdder } n a b \text{ cin s c } \\
= \text{ let } SUM = val a n + val b n + bv \text{ cin}  \\
\text{ and } MAX = 2^{n+1}  \\
\text{ in } val s n = (SUM \mod MAX) \land  \\
\text{ c } = (SUM \geq MAX)
\]

Notice that the specification makes no mention of any internal components nor wiring. The specification is clear and succinct precisely because it abstracts away information on the internal structure of the device.

Here are the specifications of some other common arithmetic subsystems:

\[
\text{nInc } n a c i n s c \\
= \text{ let } SUM = val a n + bv \text{ cin}  \\
\text{ and } MAX = 2^{n+1}  \\
\text{ in } val s n = (SUM \mod MAX) \land  \\
\text{ c } = (SUM = MAX)
\]

\[
\text{ntosComp } n a c i n s c \\
= \text{ let } SUM = val a n + bv \text{ cin}  \\
\text{ and } MAX = 2^{n+1}  \\
\text{ in } val s n = (MAX - SUM \mod MAX) \land  \\
\text{ c } = (SUM = 0)
\]

Notice that not only do these specifications follow the same pattern, but also that this is the same pattern as that of the arithmetic gates. This leads to the interesting observation that if we stick to this template we can easily predict the specification of a subsystem composed from one cell, by taking the specification of the cell, (except for the carry lines) systematically changing terms like \(bv\) to \(val\) \(s\) \(0\), and then substituting \(val\) \(s\) \(n\) for \(val\) \(s\) \(0\) and \(2^{n+1}\) for 2 wherever they occur. It even leads to the hope that if we have a specification and a theorem for a 1-bit arithmetic device in terms of \(bv\) then we may be able to infer a specification and a correctness statement for a row of such devices in terms of \(val\) \(n\). At the very least, when we have a specification of a 1-bit device, writing the specification for the \(n\)-bit device is mechanically easy.

Notice also that, with the right abstractions (\(bv\) and \(val\)), specifications need not balloon in size as we move from gates to subsystems.
5.1.2 Defining implementations

An implementation definition tells you how a device is constructed by stating its constituent parts and how they are wired together. We show how this is done via examples.

**Example 5.1.4 Implementation of mux2**

There are several ways to implement a mux2 gate. One way, which wires an inverter and three nand2 gates together, is depicted in figure 5.6.

![Figure 5.6 mux2 implementation](image)

The separate definitions of the four constituents are:

\[
\begin{align*}
\text{inv sel selbar} & \quad \text{nand2 a selbar } p \\
\text{nand2 sel selbar} & \quad \text{nand2 p q z} \\
\text{nand2 sel b q} & \quad \text{nand2 q p z}
\end{align*}
\]

where we have instantiated one inverter and three nand2 gates by supplying actual line names to the template definitions. To implement a mux2 gate, we certainly require that four relations, namely

1. that selbar is in the inverse relation to sel
2. that p is in the nand2 relation to a and selbar
3. that q is in the nand2 relation to sel and b, and
4. that z is in the nand2 relation to p and q

hold at the same time so it is fair to use \( \wedge \) as the composition operator. As a first attempt at supplying a complete definition of this implementation we try

\[
\text{mux2_imp sel a b z} = \text{inv sel selbar} \wedge \text{nand2 a selbar p} \wedge \\
\text{nand2 sel b q} \wedge \text{nand2 p q z}
\]
but this is not satisfactory because \texttt{selbar}, \texttt{p}, and \texttt{q} are not defined locally. Again

\begin{align*}
\texttt{mux2.imp} \ & \texttt{sel} \ a \ b \ \texttt{selbar} \ p \ q \ z \\
= \ & \texttt{inv \ sel} \ \texttt{selbar} \ & \texttt{nand2} \ a \ \texttt{selbar} \ p \ & \texttt{nand2} \ b \ q \ & \texttt{nand2} \ p \ q \ z
\end{align*}

is unsatisfactory as it treats the internal lines \texttt{selbar}, \texttt{p}, and \texttt{q} in exactly the same manner as the external wires as arguments to the definition. Values on internal lines are not freely available as are the values on lines \texttt{sel}, \texttt{a}, \texttt{b}, and \texttt{z}. They are constrained to be in a certain relation with the inputs to that device. For instance (see figure 5.6, we know that the \texttt{selbar} value emerging from the inverter gate carries the value \texttt{~sel}).

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{figure5.7.png}
\caption{Equivalent \texttt{mux2} implementation}
\end{figure}

Given this constraint, the circuit shown in figure 5.7 is equivalent to the circuit shown in figure 5.6 when it has one extra constrained input as shown.

Similarly given that \texttt{selbar} = \texttt{~sel}, the value on the \texttt{p} line emerging from the top \texttt{nand2} gate carries the value \texttt{~(a \ \texttt{selbar})} and the value on the \texttt{q} line emerging from the bottom \texttt{nand2} gate is \texttt{~(sel \ \texttt{b})}. See figure 5.8.

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{figure5.8.png}
\caption{Another equivalent \texttt{mux2} implementation}
\end{figure}

What we need is a notation that reflects these facts together with a logical inference rule which allows us to carry out these substitutions correctly.
We already have such a rule: it is \texttt{EXISTS\_ELIM} (and stronger variants). In order to employ this rule correctly in HOL, we existentially quantify all its purely internal wires when we define an implementation. The definition of \texttt{mux2\_imp} becomes

\[
\begin{align*}
mux2\_imp & \ sel \ a \ b \ z \\
\quad & = \exists \ \text{selbar} \ p \ q . \\
& \quad \ inv \ sel \ \text{selbar} \ \land \ \text{nand2} \ a \ \text{selbar} \ p \ \land \\
& \quad \ \text{nand2} \ sel \ b \ q \ \land \ \text{nand2} \ p \ q \ z
\end{align*}
\]

which is read as: “this mux2 implementation has 4 external ports (the fact that \texttt{sel}, \texttt{a} and \texttt{b} are input ports, and \texttt{z} is an output port can be inferred), and is built by wiring together one inverter and three \texttt{nand2} gates using \texttt{selbar}, \texttt{p}, and \texttt{q} as internal lines”.

Why is this what we want? When we carry out a verification, we first unfold the implementation definition by substituting for the constituent parts, and then use \texttt{EXISTS\_ELIM} to remove first \texttt{selbar} and then \texttt{p} and \texttt{q}.

\[
\begin{align*}
mux2\_imp & \ sel \ a \ b \ z \\
\quad & \rightarrow \\
\quad & \exists \ \text{selbar} \ q \ r . \\
& \quad \ inv \ sel \ \text{selbar} \\
& \quad \ \land \ \text{nand2} \ a \ \text{selbar} \ p \\
& \quad \ \land \ \text{nand2} \ sel \ b \ q \\
& \quad \ \land \ \text{nand2} \ p \ q \ z \quad \ \text{DEFINITION}
\end{align*}
\]

\[
\begin{align*}
\quad & \rightarrow \\
& \exists \ \text{selbar} \ q \ r . \\
& \quad \ \text{selbar} = \ \neg \text{sel} \quad \ \text{definition of inv} \\
& \quad \ \land \ p = \ \neg (a \land \neg \text{sel}) \quad \ \text{definition of nand2} \\
& \quad \ \land \ q = \ \neg (sel \land b) \quad \ \text{definition of nand2} \\
& \quad \ \land \ z = \ \neg (p \land q) \quad \ \text{definition of nand2} \quad \ \text{SUBSTITUTION}
\end{align*}
\]

\[
\begin{align*}
\quad & \rightarrow \\
& \exists \ q \ r . \\
& \quad \ p = \ \neg (a \land \neg \text{sel}) \\
& \quad \ \land \ q = \ \neg (sel \land b) \\
& \quad \ \land \ z = \ \neg (p \land q) \quad \ \text{EXISTS\_ELIM on selbar}
\end{align*}
\]

\[
\begin{align*}
\quad & \rightarrow \\
& \quad \ \text{EXISTS\_ELIM on p and q} \\
& \quad \ z = \ \neg (\neg (a \land \neg \text{sel}) \land \neg (sel \land b))
\end{align*}
\]

After we have removed the hidden lines, we have an expression for the output line in terms of the inputs which can be simplified by laws of logic. Notice the strategy here: we remove the hardware components with their definitions (logical relations) and then the hidden lines. What we are left with is pure logic with no explicit “hardware” at all.
Example 5.1.5 Implementation of a word ripple carry adder

The implementation consists of \( n+1 \) full adders wired together with internal lines labelled \( q_1 \) through \( q_{n-1} \) as shown in figure 5.9.

![Diagram of a word ripple carry adder](image)

Figure 5.9 n-bit adder implementation

The implementation is defined by primitive recursion: a 0-bit adder is a single 1-bit full adder, and an \((n+1)\)-bit adder is a constructed by composing an \( n \)-bit adder and a full adder as in figure 5.10.

![Diagram of recursive construction of an n+1-bit adder](image)

Figure 5.10 Recursive construction of an \( n+1 \)-bit adder

In the inductive case the hidden wire is named \( q \). It is the carry-out of the \( n \)-bit adder and the carry-in of the full adder.
The fact that we define regular structures in this way simplifies the verification task since there is a nice match between recursive definitions and induction. Consider the goal \( \forall n. \mathrm{nAdder\_imp} n a b cin s c = \mathrm{nAdder\_spec} n a b cin s c \) with \( \mathrm{nAdder\_imp} \) defined by primitive recursion. An inductive proof requires that we prove

**Base case.** \( \mathrm{nAdder\_imp} 0 a b cin s c = \mathrm{nAdder\_spec} 0 a b cin s c \) which is trivial since

\[
\begin{align*}
\mathrm{nAdder\_imp} 0 a b cin s c &= \text{full\_adder\_imp} a0 b0 cin s0 c \\
\mathrm{nAdder\_spec} 0 a b cin s c &= \text{full\_adder\_spec} a0 b0 cin s0 c
\end{align*}
\]

and we will have a proof for the \texttt{full\_adder}.

**Inductive case.** We have to show that

\( \mathrm{nAdder\_imp} (n+1) a b cin s c = \mathrm{nAdder\_spec} (n+1) a b cin s c \) under the assumption that

\( \mathrm{nAdder\_imp} n a b cin s c = \mathrm{nAdder\_spec} n a b cin s c \).

We proceed as follows:

1. rewrite with the definition of \( \mathrm{nAdder\_imp} (n+1) a b cin s c \). The goal becomes

\[
? q. \mathrm{nAdder\_imp} (n+1) a b cin s q \\
\text{\textbackslash \textbackslash \mathrm{full\_adder\_imp} a(n+1) b(n+1) q s(n+1) c} \\
= \mathrm{nAdder\_spec} (n+1) a b cin s c
\]

2. rewrite with the induction hypothesis and the goal becomes

\[
? q. \mathrm{nAdder\_spec} (n+1) a b cin s q \\
\text{\textbackslash \textbackslash \mathrm{full\_adder\_imp} a(n+1) b(n+1) q s(n+1) c} \\
= \mathrm{nAdder\_spec} (n+1) a b cin s c
\]

3. rewrite with the correctness statement for \texttt{full\_adder\_spec} and the goal becomes
The goal is now simplified to specifications on the left and on the right. The rest is up to our expertise in HOL.

5.2 Verification of circuits

In this final section we introduce the basic notions of circuit verification. What we have to do is show that the implementation meets the specification over all inputs and outputs using the backward proof style. The style of a backward proof is straightforward: remove quantified variables, substitute the definitions of the implementation and its components and the specification, remove all hidden lines using some variant of \texttt{EXISTS\_ELIM}. What will then be left is an equality whose left part and right part are conjunctions of expressions for each output line in terms of input variables. The left hand side is derived from the definition of the implementation, and the right hand side is derived from the definition of the specification. We then use standard techniques of mathematical logic to show that we have a tautology.

Example 5.2.6 $\forall a\ b\ z . \ mux_2\_imp\ a\ b\ z = mux_2\_spec\ a\ b\ z$

For convenience, we repeat the definitions of the specification, and the implementation.

\[
\begin{align*}
mux_2\_spec\ &sel\ a\ b\ z = (z = \neg sel \Rightarrow a \mid b) \\
mux_2\_imp\ &sel\ a\ b\ z \\
&= \ ? selbar\ p\ q . \\
&\quad (inv\ sel\ selbar) \ /\n\quad (nand2\ a\ selbar\ p) \ /\n\quad (nand2\ sel\ b\ q) \ /\n\quad (nand2\ p\ q\ z)
\end{align*}
\]

Here are the main steps in the proof:

1. first set the goal.
2. remove the universal quantification.
3. rewrite with the definitions of `mux2_imp` and `mux2_spec`.

4. on the left hand side we now have a conjunction of “simpler” hardware components, each of which is a primitive. We rewrite with their specifications. At this stage all the hardware has gone and we are in the realm of mathematical logic.

5. remove the hidden lines using `EXISTS_ELIM`, or stronger variants.

6. use the rules and methods of mathematical logic to prove or disprove the goal.

In the presentation below, each step in the proof is justified by appeal to some proof rule.

**GOAL:**

```
! sel a b z . mux2_imp sel a b z = mux2_spec sel a b z
```

~ eliminating the quantified variables

```
mux2_imp sel a b z = mux2_spec sel a b z
```

~ rewriting with the definitions of `mux2_imp` and `mux2_spec`

```
? selbar p q .
  (inv sel selbar) /
  (nand2 a selbar p) /
  (nand2 sel b q) /
  (nand2 p q z) = z = (~ sel => a | b)
```

~ rewriting with the definitions of `inv` and `nand2`

```
? selbar p q .
  selbar = ~ sel /
  p = ~(a \ selbar) /
  q = ~(sel \ b) /
  z = ~(p \ q) = z = (~ sel => a \ b)
```

~ eliminating `selbar` by using `EXISTS-ELIM`

```
? p q .
  p = ~(a \ ~ sel) /
  q = ~(sel \ b) /
  z = ~(p \ q) = z = (~ sel => a \ b)
```

~ eliminating `p` and `q` using `EXISTS-ELIM`

```
(z = ~(~(a \ ~ sel)) \ ~(sel \ b))) = (z = (~ sel => a \ b))
```
5.2. VERIFICATION OF CIRCUITS

\[ (z = \neg (a \lor \neg sel) \lor (sel \lor b)) = (z = (\neg sel \Rightarrow a \lor b)) \]

\[ (z = (a \lor \neg sel) \lor (sel \lor b)) = (z = (\neg sel \Rightarrow a \lor b)) \]

\[ \neg \text{ which may be verified by casing on sel} \]

\[ \text{case } sel = T: \ (z = F \lor b) = (z = b) \]

\[ \text{case } sel = F: \ (z = a \lor F) = (z = a) \]

and both cases simplify to a tautology. \qed

Notice that we have achieved our result by a mixture of syntactic manipulation and semantic interpretation (bool casing).

**Example 5.2.7 nBuffer.**

Our second example verifies a line of inverters as shown in figure 5.11. Until now, we have specified broadside devices where each component has input and output wires which can be seen from the outside. In this subsystem, the signal traverses the whole subsystem before being emitted and only two lines can be seen from the outside.

![Figure 5.11 nInv implementation](image)

Given the definition of the auxiliary function `even`

\[
\text{even } 0 = T \\
\text{even } \text{(Suc } n) = \neg (\text{even } n)
\]

the specification of `nBuffer` is

\[ ! n i z . \ \text{nBuffer_spec } n \ i \ z = (z = \text{even } n \Rightarrow i \lor \neg i) \]

Note that \( z = i \) for the zero-order device `nBuffer_spec 0 i z`.

The implementation is easy to define when we use primitive recursion. We take care that the physical layout matches induction index. Thus to implement this device, we just wire \( n \) unit inverters together in ripple fashion. The zero-order case is just a straight wire.
Informal proof of nBuffer

Proofs of regular sub-systems (where the implementation is defined by primitive recursion) use induction. The proof is split into a base case and an induction step.

Case \( n = 0 \):

\[
! i z . \text{nBuffer\_imp} \; 0 \; i z = \text{nBuffer\_spec} \; 0 \; i z
\]

\( \vdash \) eliminating the quantifiers and rewriting with the definitions of \text{nBuffer\_imp} \; 0 and \text{nBuffer\_spec} \; 0

\( (z = i) = (z = \text{even} \; 0 = \rightarrow i \mid i) \)

and a further rewrite with \text{even} leaves us with a tautology.

Case \( n + 1 \):

In induction proofs, the induction assumption is shown in square brackets underneath the goal.

\[
! i z . \text{nBuffer\_imp} \; (n+1) \; i z = \text{nBuffer\_spec} \; (n+1) \; i z
\]

\( \vdash \) Step 1: eliminate the generalisations

\[
\text{nBuffer\_imp} \; (n+1) \; i z = \text{nBuffer\_spec} \; (n+1) \; i z
\]

\( \vdash \) Step 2: the right hand side of the goal is already “mathematical”. Our job is to work the left hand side into formal mathematics. Rewrite with the (primitive recursive) definition of \text{nBuffer\_imp} \; (n+1)

\[
? q. (\text{nBuffer\_imp} \; n \; i q \; / \; \text{inv} \; q \; z) = \text{nBuffer\_spec} \; (n+1) \; i z
\]

\( \vdash \) Step 3: use the induction hypothesis to replace \text{nBuffer\_imp} \; n by \text{nBuffer\_spec} \; n

\[
? q. (\text{nBuffer\_spec} \; n \; i q \; / \; \text{inv} \; q \; z) = \text{nBuffer\_spec} \; (n+1) \; i z
\]

\( \vdash \) Step 4: rewrite with the definition of inv
5.2. VERIFICATION OF CIRCUITS

? q. (nBuffer_spec n i q \( z = \text{even} n \)) = nBuffer_spec (n+1) i z 
[ ! i z . nBuffer_imp n i z = nBuffer_spec n i z ]

\( \sim \) Step 5: rewrite with the definition of nBuffer_spec

\( (? q . q = \text{even} n \Rightarrow i | \sim i /\ z = \sim q) = (z = \text{even} (n+1) \Rightarrow i | \sim i) \)
[ ! i z . nBuffer_imp n i z = nBuffer_spec n i z ]

\( \sim \) Step 6: eliminate the hidden line q

\( (z = \sim (\text{even} n \Rightarrow i | \sim i)) = (z = \text{even} (n+1) \Rightarrow i | \sim i) \)
[ ! i z . nBuffer_imp n i z = nBuffer_spec n i z ]

\( \sim \) Step 7: substitute for even \((n+1)\)

\( (z = \sim (\text{even} n \Rightarrow i | \sim i)) = (z = \sim (\text{even} n) \Rightarrow i | \sim i) \)
[ ! i z . nBuffer_imp n i z = nBuffer_spec n i z ]

\( \sim \) Step 8: if this is not clearly true, try cases on even n

case even n = T: \( (z = \sim (T \Rightarrow i | \sim i)) = (z = \sim (T) \Rightarrow i | \sim i) \)
which reduces to \( (z = \sim i) = (z = \sim i) \)
case even n = F: \( (z = \sim (F \Rightarrow i | \sim i)) = (z = \sim (F) \Rightarrow i | \sim i) \)
which reduces to \( (z = \sim (\sim i)) = (z = i) \)
both of which are clearly true.

\( \square \)
**EXERCISES 5**

**Exercise 5.1** Here are the specifications of several other common logic gates. Sketch the gates from these specifications, identify the input ports and the output ports, and type them. Give truth tables for each gate and check that they are equivalent to the specification.

\[ \forall a \ b \ z . \text{ nand } a \ b \ z = (z = \neg (a \land b)) \]
\[ \forall a \ b \ z . \text{ nor } a \ b \ z = (z = \neg (a \lor b)) \]
\[ \forall a \ b \ z . \text{ and } a \ b \ z = (z = a \land b) \]
\[ \forall a \ b \ z . \text{ or } a \ b \ z = (z = a \lor b) \]
\[ \forall a \ b \ z . \text{ xor } a \ b \ z = (z = \neg (a = b)) \]
\[ \forall a \ b \ z . \text{ xnor } a \ b \ z = (z = (a = b)) \]

**Exercise 5.2** Specify the following circuits: half adder, one’s complementer, two’s complementer; subtracter, multiplier, divider, encoder, decoder, comparator.

**Exercise 5.3** Specify the following n-bit arithmetic circuits: incrementer, decrementer, subtracter, one’s complementer, two’s complementer, multiplier, divider.

**Exercise 5.4** Specify a collection of registers and shifters.

**Exercise 5.5** Define the implementations of all the circuits specified above.

**Exercise 5.6** Verify your half adder, full adder, and adder sub-system implementations. Warning: the verification of the adder sub-system will probably take several pages, but it is a good test of skill and perseverance.

**Exercise 5.7** As we have seen, specifications are mathematical in nature, undaunted by implementation details, and hence much nicer to work with than the latter. It is often possible to infer useful theorems from specifications. Sometimes these can be used to simplify proofs, sometimes they can be used to check out the consequences of specifications. To give the idea, here are two simple theorems that enable us to visualise an adder and its carry-out as “an adder of the next size”.

a. Prove that the inputs and outputs of a full adder are in the relation
\[ bv s + 2 \times (bv cout) = \text{sum } a \ b \ cin \text{ where } \text{sum } a \ b \ cin = bv a + bv b + bv cin. \]

b. Prove that the inputs and outputs of an nAdder are in the relation
\[ \text{val } s \ n + 2^{n+1} \times (bv cout) = SUM n a \ b \ cin \text{ where } \text{SUM } n a \ b \ cin = \text{val } a \ n + \text{val } b \ n + bv cin. \] Use induction.
Chapter 6

Uses and limitations of verification

Today’s VLSI designs are complicated pieces of hardware containing tens or even hundreds of thousands of gates. To bridle complexity, the design of such a complex system is tackled by some divide-and-conquer approach. One common strategy is to elaborate a design hierarchically. A hierarchy may be built from the top down, from the bottom up, or by a mixture of these approaches doing whatever is locally more suitable. In hierarchical design we make sure that each level of the design elaboration is correct (that the behaviours of the children of a node compose to their parent’s behaviour) before proceeding with their elaborations. We continue in this way until we have reached the level of primitives that are known. Then by deductive reasoning, we have shown the correctness of the whole design. In this chapter, we look at the benefits of hierarchical design, compare the simulation and formal approaches to verification, and comment on the limitations of the approach.

6.1 Hierarchies

A hierarchical design results in a tree structure with the children of each node being an elaboration of their parent. Figure 6.1 shows the elaboration of a 4-bit ripple carry adder down to the gate level. (Only the second branch of the hierarchy is developed in full in figure 6.1 — the other three branches are obviously going to be the same.) This 4-bit adder is first partitioned into four 1-bit full adders, each of which is constructed from two half adders and an and gate. Each half adder is constructed from an xor gate and an or gate. Here we are going to assume that xor, and, or and inverter gates are available to us as library primitives. That being so, our design elaboration is complete since we now have the entire design expressed in terms of known elements.

The bonuses of working with a hierarchy are: (i) we work with smaller problems which may be considered separately; (ii) where parts are replicated, we need do the verification only once; (iii) because we are working
on only a portion of a design at a time, mismatches between the implementation and the specification pinpoint design errors; and (iv) the effects of replacing a part, or parts, of a design are limited. If we don’t work with a hierarchy, then we would have to verify a flattened design consisting of 20 primitive gates (exclusive ors, ands, and ors) with 9 input lines, 5 output lines, and 15 internal lines. Importantly, we would spend much of our time repeating work already done, since the sub-patterns are repeated several times. Said in another way, to take full advantage of working with a hierarchy, we must be able to carry out verifications at each of its levels. I.e. we need a notation that can cope with transistors, gates, subsystems and architectures, and we must be able to carry out complete certifications at each and every one of these levels in that notation.

6.2 Simulation

The most widely used method for verifying hardware designs is simulation. To verify the top level of this design elaboration, we would run a battery of tests through a model of the implementation trying all possible input variations and checking to see that the design meets the specification. If the results agree in all cases then we have verified this level of the hierarchy. If not we have to change the design (or the specification!) until they do match. In order to verify the complete design, we have to repeat the exercise at each level of the hierarchy, until every non-primitive gate (node in the
hierarchy tree) has been verified. Then we may infer that the design meets its specification.

Simulation is best done by a computer program — the one below will carry out one test of the design of an adder subsystem of arbitrary size at the top level of the hierarchy. To validate our design for a 4-bit adder exhaustively at this level, we would set \( n = 3 \) (the bits are numbered 0–3) and run through \( 2^9 \) test patterns. The program is intended to be clear rather than smart.

\[
\% \text{ INITIALISE the run by reading in values for the } \%
\% \text{ size of the adder (n), the vectors to be added } \%
\% (a and b), and the carry-in bit (cin) } \%
\]

\[
\text{function } \text{bv}(b) = \text{if } b \text{ then } 1 \text{ else } 0; \\
\text{function } \text{val}(n, a) = \\
\quad \text{if } n = 0 \text{ then } \text{bv}(a[0]) \text{ else } \text{val}(n-1, a) + (\text{EXP } 2 \times n) \times \text{bv}(a[n]) \\
\]

\[
n := \text{read}; a := \text{read}; b := \text{read}; \text{cin} := \text{read}; \\
\text{max} := (\text{EXP } 2 \times (n+1)); \\
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
\%
The major problem with simulation is that it is so time consuming that it cannot be done exhaustively for even moderately large designs. Exhaustive simulation at the top level for an $n$-bit adder requires $2^n$ values of $a$, $2^n$ values of $b$, and by 2 cases for $cin$ and $2^{2n+1}$ grows large very quickly with increasing $n$. The number of necessary cases may be drastically reduced by choosing test vectors wisely, but this is a tricky business and is usually not attempted with rigour — rule of thumb prevails. As a result, most of today’s chips are only partially validated since only a percentage of the set of possible inputs is tried. This inevitably results in some production chips being flawed.

### 6.3 Simulation vs formal verification

It is instructive to compare our formal specifications for the 1-bit full adder

<table>
<thead>
<tr>
<th>Formal specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>full_add $a \ b \ cin \ s \ c$</td>
</tr>
<tr>
<td>$bvs = \sum \mod 2 \land \ c \ = \ \lfloor \sum \geq 2 \rfloor$</td>
</tr>
<tr>
<td>where $\sum = bv a + bv b + bv cin$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Simulation model</th>
</tr>
</thead>
<tbody>
<tr>
<td>$sum : = \bv[a[k]] + \bv[a[k]] + \bv[q[k]]$</td>
</tr>
<tr>
<td>$s [k] : = \text{if} \ \sum &lt; 2 \ \text{then} \ \sum \ \text{else} \ \sum - 2$</td>
</tr>
<tr>
<td>$q [k] : = \lfloor \sum \geq 2 \rfloor$</td>
</tr>
</tbody>
</table>

#### Table 6.1 Formal and simulation specifications of the 1-bit full adder

and for the word adder with the code used in the simulation model. As with the formal specification, the simulation model also specifies the value to be seen on each output line in terms of current inputs. Furthermore, these are mere notational variations on our formal specifications. So clearly, simulation programs use formal specifications. Further the level of detail is the same in both cases, i.e. each use the same underlying model. The difference lies in how we use these specifications. Simulations run many individual tests to cover all input possibilities. Formal proofs take the same model and the same specifications as are used in the simulation model and manipulate the latter formally to prove the correctness of the design elaboration. Full coverage, that is correctness over all input values, comes...
6.4 Limitations to verification

The implementation meets its specification by completely formal methods, but formal verification is not a universal panacea. Besides being hard and painstaking, the approach has its limitations. Here are some of the drawbacks.

---

<table>
<thead>
<tr>
<th>Formal specification</th>
</tr>
</thead>
</table>
| \[\text{nAdder } n \ a \ b \ \text{cin} \ s \ c \]  
| \[= \quad \begin{array}{l}
\text{val n} = \text{sum} \mod \text{max} \\
\text{c} = \text{if } \text{sum} < \text{max} \text{ then SUM else SUM-max;}
\end{array}\]  
| where \text{sum} = \text{val n a} + \text{val n b} + \text{by cin}  
| and \text{max} = 2^{n+1}  

<table>
<thead>
<tr>
<th>Simulation model</th>
</tr>
</thead>
</table>
| \[\begin{array}{l}
\text{sum} := \text{val}(n, a) + \text{val}(n, b) + \text{by}(\text{cin}); \\
\text{s} := \text{if } \text{SUM} < \text{max} \text{ then SUM else SUM-max;}
\end{array}\]  
| \[\text{c} := \text{SUM} \mod \text{max};\]  

Table 6.2 Formal and simulation specifications of the word adder
• what we will be proving is that a hardware design meets its specification. We cannot prove that a design is correct because there may be a gap between what the customer wants and the top-level formal specification. This can never be bridged by formal reasoning (neither can it be bridged by informal reasoning, so we are no worse off). Confidence in the correctness of a specification can be gained exercising it with a high level (symbolic?) simulator, formally deriving consequences (theorems) from it (the style we prefer), and by subjecting it to public scrutiny (hence the requirement for succinct and clear formal specifications).

• Defining an implementation which is expected to meet the specification. To define an implementation, we have to list its constituent parts and state how they are wired together. Thinking of an implementation is hard — defining it is easy.

• Verifying that the design meets the implementation for all input sequences. Formal methods use the same models and the same specifications as are used in simulations (at the switch level and above). The essence of good VLSI design is regularity and the advantage of formal verification is that one can use induction to verify a regular systems. So for example, the n-bit adder, can be verified in two (albeit harder) steps instead of an exponential number of small ones. The verification would be carried out for the general case and then be specialised to, say, n = 32.

• it is important to realise that specifications and implementation definitions are based upon a model of hardware, and that correctness proofs are valid to within the tolerance of that model. For example in this book, we only use strong signals and our model cannot properly describe or be used to verify any circuits whose success depends upon more variety in signal strength. As a second example, our proofs on sequential circuits implicitly assume that suitable clocking rates are known. Our models are probably adequate for gate array chips, but are not fine enough for full custom VLSI design. However, work on the detailed modelling of circuits is going on, see for example [39, 52, 86, 115]. For recent work on clock primitives and design style axiomatisation see [34, 35, 55, 56, 71, 84].

• a verification is only as strong as its weakest link. No step may be omitted and no section of a proof may be skipped. Practical experience has shown that hand proofs are less trustworthy than machine-assisted (or machine-checked) proofs [28]. So although it may take
longer to complete a formal verification, the effort is worthwhile. Unfortunately this raises the question Is your proof-checker verified? — and none of them are. All we can say in our defence is that, over the years, several proof-checkers have been found to be reliable and robust. Still, machine assisted verifications should be treated as being probably correct rather than guaranteed correct. Interesting and important references on this topic are papers by Cohn [26, 27, 28] and by Rushby and von Henke [98].

In passing, we note another reason for wanting machine assisted proofs of hardware designs is the possibility that then emerges of transforming the results of the verification effort into layout [14, 103].

- the methodology, tools, and experience are not yet there, i.e. the subject is still in its infancy. There is a strong need for libraries of specifications to be established, and for large case studies to be published so that we can work towards establishing a robust and reliable technology and automating (part of) it. This is starting to happen — see for example, Computational Logic’s continuing system-level verification work [5, 6, 7, 82, 83, 116], Chin’s work on arithmetic units [18, 19, 20, 21, 22], Cohn’s work on VIPER [26, 27], the work on SECD [9, 47, 48, 49], Hunt’s verification of FM8501 [60], Mike Gordon’s classic computer proof [42] and follow-up work by Joyce [65, 66, 67], SRI’s work on clocks and fault-tolerant systems [96, 97, 98, 100], Shankar’s tour de force [99], and Windley’s PhD and follow-up work [111, 112, 113].

Except for the last point, the negatives are of course common to all VLSI design verification tools, including simulation. Despite these current drawbacks, formal verification remains an approach with a promising future. Formal verification should be considered as another weapon in the armoury of hardware designers, particularly useful for reasoning about system timing, for showing the correctness of regular systems, and for conducting proofs of functionality at the subsystem level and above. See [30, 31] for a recent survey of industrial applications of formal methods.
Part III

Starting with the HOL system
A basic library of gates

The basic HOL system contains no primitives aimed specifically at supporting work in hardware verification. Thus our first task is to establish a collection of basic building blocks from which we can construct and verify more interesting circuits. The HOL library construct is called a theory. A theory may contain definitions (circuit specifications and descriptions of implementations) and theorems (typically, correctness statements and auxiliary facts). Theories are organized in a hierarchy, so that constants, definitions, and theorems of a parent theory are inherited by the child theory.

In the course of a typical session with HOL, we first open a new working theory and enter “draft” mode. We then declare a number of existing theories to be parents, whose definitions and theorems will be used as building blocks in the verification of more complex hardware designs. New definitions and theorems will be saved in the working theory as they are derived. At the end of the session, we save all our results when we close the working theory, leaving draft mode. Outside of draft mode, we cannot make new definitions or declare new parents, but we can still prove and save theorems.

We open a fresh theory by calling `new_theory`

```ml
# new_theory;;
- : (string -> void)
```

For example `new_theory "gates"` causes a new theory file called `gates.th` to be opened in your directory into which HOL will store suitably “compiled” versions of the definitions we make and the theorems we prove. Since `new_theory` is an ML function, any call must return a value. The value returned is `()` which has type `void`.

The general format of each new definition in HOL is

```ml
let item1 = new_definition ("item1", "item3 args = body");
```

where `new_definition` is an ML function.

```ml
# new_definition;;
- : ((string # term) -> thm)
```
new_definition takes a string ‘item$3$’ and a term “item$3$ args = body” as arguments. It interprets the term as a HOL definition and stores it away in the current theory under its string name (here ‘item$3$’). new_definition returns the newly generated theorem as its result. The result may be bound to a local ML variable (here item$1$). Although the names item$1$, item$2$, and item$3$ may be different, it gets rather confusing if they are. We will always make them textually the same.

```
#let inv = new_definition
   ('inv',
    "inv i out = (out = ~i)"));
inv = |- !i out. inv i out = (out = ~i)
```

Notice that the free variables in the definition (i and out in the example above) are automatically generalised in the returned theorem.

When we prove substantial theorems in the text, we will generally use the backward proof technique and make use of prove_thm. Here is a short example. (CONJ_ASSOC is in fact a built-in HOL theorem.)

```
#prove_thm();
- : ((string # term # tactic) -> thm)

#let CONJ_ASSOC = prove_thm
   ('CONJ_ASSOC',
    " ! a b . (a \ (b \ / c)) = ((a \ b) \ / c)",
    REPEAT GEN_TAC
    THEN BOOL_CASES_TAC "a:bool"
    THEN REWRITE_TAC []
    );
CONJ_ASSOC = |- !a b. a \ / b \ / c = (a \ / b) \ / c
```

prove_thm takes a string (here 'CONJ_ASSOC') under which name the theorem returned is saved in the current theory; a goal, which is a term representing the fact we are trying to establish, here

" ! a b . (a \ (b \ / c)) = ((a \ b) \ / c)"

and a tactic that will prove the theorem, here

```
REPEAT GEN_TAC THEN BOOL_CASES_TAC "a:bool" THEN REWRITE_TAC []
```

As distinct from free variables in definitions, free variables in goals are NOT automatically generalised.
7.1 A theory of basic gates

This session starts with us entering HOL, opening a new theory called gates and entering a number of basic definitions. The resulting theorems are stored away in the theory gates and echoed back to us. Especially in the early stages, we are rather lavish with parentheses in order to make our intentions quite clear. Just as in ML, the rest of a line after a \% is taken as a comment.

```
#new_theory 'gates';
() : void

#let inv = new_definition
('inv', "inv i out = (out = ~i)" );
inv = ~i out. inv i out = (out = ~i)

#let nand2 = new_definition
('nand2', "nand2 a b out = (out = ~\(a \land b\))")
and nand3 = new_definition
('nand3', "nand3 a b c out = (out = ~\((a \land b) \land c\))")
and nand4 = new_definition
('nand4', "nand4 a b c d out = (out = ~\((a \land b) \land c \land d\))")
and nand5 = new_definition
('nand5', "nand5 a b c d e out = (out = ~\((a \land b) \land c \land d \land e\))");
nand2 = ~a b c out. nand2 a b out = (out = ~\(a \land b\))
nand3 = ~a b c out. nand3 a b c out = (out = ~\((a \land b) \land c\))
nand4 = ~a b c d out. nand4 a b c d out = (out = ~\((a \land b) \land c \land d\))
nand5 = ~a b c d e out.
   nand5 a b c d e out = (out = ~\((a \land b) \land c \land d \land e\))

#let nor2 = new_definition
('nor2', "nor2 a b out = (out = ~\(a \lor b\))")
and nor3 = new_definition
('nor3', "nor3 a b c out = (out = ~\((a \lor b) \lor c\))")
and nor4 = new_definition
('nor4', "nor4 a b c d out = (out = ~\((a \lor b) \lor c \lor d\))")
and nor5 = new_definition
('nor5', "nor5 a b c d e out = (out = ~\((a \lor b) \lor c \lor d \lor e\))");
nor2 = ~a b c d e out. nor2 a b out = (out = ~\(a \lor b\))
nor3 = ~a b c d e out. nor3 a b c out = (out = ~\((a \lor b) \lor c\))
nor4 = ~a b c d e out. nor4 a b c d out = (out = ~\((a \lor b) \lor c \lor d\))
nor5 = ~a b c d e out. nor5 a b c d e out = (out = ~\((a \lor b) \lor c \lor d \lor e\))
```

In the definitions above, HOL will correctly infer that all the input and output ports are boolean. If you volunteer type information, then HOL will check its consistency and tell you when you get it wrong. Explicit
typing information can be used as an added documentation aid. We will feel free to decorate our definitions with explicit types when we feel it will aid understanding.

Another handy HOL function is called `print_theory`. Supplied with the (string) name of a theory, it will list the contents of the theory.

```haskell
# print_theory;;
- : (string -> void)

# print_theory 'gates';;
The Theory gates
Parents -- HOL
Constants --
  inv "bool -> (bool -> bool)"
  nand2 "bool -> (bool -> (bool -> bool))"
  nand3 "bool -> (bool -> (bool -> (bool -> bool)))"
  nand4 "bool -> (bool -> (bool -> (bool -> (bool -> bool))))"
  nand5 "bool -> (bool -> (bool -> (bool -> (bool -> (bool -> bool))))")
  nor2 "bool -> (bool -> (bool -> bool))"
  nor3 "bool -> (bool -> (bool -> (bool -> bool)))"
  nor4 "bool -> (bool -> (bool -> (bool -> (bool -> bool))))"
  nor5 "bool -> (bool -> (bool -> (bool -> (bool -> (bool -> bool))))")"

Definitions --
  inv |\!i out. inv i out = (out = \!i)
  nand2 |\!a b out. nand2 a b out = (out = \!(a \& b))
  nand3 |\!a b c out. nand3 a b c out = (out = \!(a \& b \& c))
  nand4 |\!a b c d out. nand4 a b c d out = (out = \!(a \& b \& c \& d))
  nand5 |\!a b c d e out.
    nand5 a b c d e out = (out = \!(a \& b \& c \& d \& e))
  nor2 |\!a b out. nor2 a b out = (out = \!(a \& b))
  nor3 |\!a b c out. nor3 a b c out = (out = \!(a \& b \& c))
  nor4 |\!a b c d out. nor4 a b c d out = (out = \!(a \& b \& c \& d))
  nor5 |\!a b c d e out.
    nor5 a b c d e out = (out = \!(a \& b \& c \& d \& e))

*************** gates ***************

() : void
```

The listing tells us that `gates` has as parent a basic theory called `HOL`. The theory `HOL` is automatically made available to every HOL session. It is where `new_theory, new_definition, print_theory, ...` are defined.

Then we find several constants generated which give the types of our (curried) definitions, one by one.
Finally the definitions we have entered are listed. Notice that each definition is saved as a theorem and becomes available as a rewriting agent since it has the form of an equation.

Finally we close down the session, saving our definitions and theorems in the current theory file with a call on close\_theory, and then quit the session.

```plaintext
#close_theory();
();
#quit();
();
Bye.
```

### 7.2 Verification of the mux2

We now start to build a respectable library of gates based upon the nine primitives already installed. We adopt the following naming convention. For each and every non-primitive device, say X, we call its specification X\_spec, its implementation X\_imp and its correctness statement X\_correct.

#### 7.2.1 Proof structure

The specification and implementation definitions for this device were given in chapter 5. Once we have the specification and a prospective implementation defined, we are in a position to attempt a proof with the HOL system. The first step is not to log on and start hacking away, but to complete an informal proof (a proof outline) by hand. This is very important for two reasons:

1. it is good to know that the goal you are attempting to prove in HOL is provable. Because machine proofs are much more detailed and messy than hand proofs, you can spend days attempting to prove the unprovable before realising that you have been wasting your time.

2. hand proofs are much shorter and quicker than machine proofs, and once you know enough HOL, you can carry out hand proofs in a way that can be mimicked in HOL. The hand proof then will guide the machine proof and tell you where you are at any given time.

We have already given an informal verification of this device in chapter 5, we get right on with the formal proof attempt. The proof is structured in 4 phases:
1. we set the goal, here \texttt{mux2$_{-}$spec} = \texttt{mux2$_{-}$spec} over all inputs and outputs.

2. we use our definitions of \texttt{mux2$_{-}$imp}, \texttt{mux2$_{-}$spec}, and \texttt{nand2} and rewrite the goal in terms of their right hand sides.

3. we eliminate hidden lines using \texttt{EXISTS\_ELIM\_TAC} a tactic corresponding to \texttt{EXISTS\_ELIM}.

4. we use techniques of mathematical logic to show that the resulting equations are tautologies under all input and output values.

7.2.2 Verification

Having entered a fresh HOL session our next step is to access the appropriate theory. In this case, we choose to \texttt{extend} the existing theory \texttt{gates} with the new definitions and theorems. We then bring down the definitions contained within the theory \texttt{gates} by a call on \texttt{load\_definitions} `gates`.

```
HOL06 Version 2.01 (SUN/AKCL), built on 4/12/92

#extend_theory;;
- : (string -> void)

#extend_theory 'gates';;
Theory gates loaded
() : void

#load_definitions;;
- : (string -> void list)

#load_definitions 'gates';;
% << *****trace omitted ***** >> %
```

Now that we have the definitions of \texttt{inv} and \texttt{nand2} to hand, we type in the definition of \texttt{mux2$_{-}$spec}

```
#let mux2$_{-}$spec = new_definition
('mux2$_{-}$spec',
 "mux2$_{-}$spec sel a b z = (z:boolean = ('sel) => a | b));
mux2$_{-}$spec = |

and “test it” by specialising the first generalised variable \texttt{sel} to \texttt{F} and then \texttt{T}
```
As expected, the output $z$ tracks $a$ in the first case, and $b$ in the second. Once satisfied we define $\texttt{mux2.imp}$. 

The first step is to set our current goal using one of two built-in functions $\texttt{set.goal}$ or $\texttt{g}$. $\texttt{set.goal}$ takes a pair as its argument, an assumption list and a term representing what we want to prove. The normal case when the assumption list is empty is catered for by $\texttt{g}$, which just takes a term and supplies an empty set of assumptions.
empty assumptions and the term as conclusion. `set_goal` takes a `goal` as argument (an assumption list \times term pair) and turns it into a new goal with the term as conclusion and the supplied assumption list as assumptions. We will use `g` nearly all the time.

HOL echoes back the current goal we are trying to prove and what assumptions we have (here, there are none).

We can now use one of HOL’s arsenal of tactics to simplify the goal. It is important to eliminate any generalisations of variables because substitutions will only be carried out on free variables. In this case we want to strip away the quantified variables! `sel a b z`. The appropriate tactic is called `GEN_TAC`.

```
#e (GEN_TAC) ;; OK ...
"!a b out. mux2_imp sel a b z = mux2_spec sel a b z"
() : void
```

**NB.** Note that when we manipulate the goal stack, HOL ensures that we do it in a secure manner. Calls on tactics must be filtered through the built-in routine called `expand` (or `e` for short). `e` is your only way of manipulating the goal.

If the tactic is accepted, HOL replies `OK` and then echoes the new goal and the current assumption list, which is still empty here. `() : void` is HOL’s response to the outer call on `expand`.

Each use of `GEN_TAC` strips away only the leading universally quantified variable. To strip off arbitrary number of generalisations, we apply `GEN_TAC` repeatedly. Any tactic may be applied zero or more times (until it fails) by prefixing it with the tactical `REPEAT`. Here `REPEAT GEN_TAC` will do the job.

```
#e (REPEAT GEN_TAC) ;; OK ...
"mux2_imp sel a b z = mux2_spec sel a b z"
() : void
```

Our next step is to replace the occurrences of `mux2_imp` and `mux2_spec` in the goal by their definitions. This we do using `PURE_REWRITE_TAC [ ... ]` which takes a list of definitions or theorems as argument and effects the rewrites with them.
We wish to simplify one level deeper and use \texttt{PURE\_REWRITE\_TAC} again to substitute for the primitive inverter and \texttt{nand2} gates.

\begin{verbatim}
#e(PURE\_REWRITE\_TAC [ inv; nand2 ]);; OK.
"(?sel p q.
    inv sel selbar /
    nand2 a selbar p /
    nand2 b sel q /
    nand2 p q z) =
    (z = (¬(sel) ⇒ a | b))"
()
\end{verbatim}

If we had seen what was coming we could have carried out the last two steps with one call on \texttt{PURE\_REWRITE\_TAC}. This will come naturally with more experience.

Our next move is to eliminate the existentially quantified variables \(p, q,\) and \(r\) through use of \texttt{EXISTS\_ELIM\_TAC}. (\texttt{EXISTS\_ELIM\_TAC} is not a built-in part of the HOL system. It is part of our standard start-up listed in appendix B).

\begin{verbatim}
#e(EXISTS\_ELIM\_TAC);; OK.
"(z = (¬(a /
    ¬ sel) /
    ¬(b /
    sel))) = (z = (¬ sel) ⇒ a | b))"
()
\end{verbatim}

Notice that in this case one application of \texttt{EXISTS\_ELIM\_TAC} eliminated all three hidden line quantities in one fell swoop. We now resort to proof by case analysis, which is an acceptable tactic when the number of cases is small. The tactic \texttt{BOOL\_CASES\_TAC} takes a boolean term as argument (here we do cases on \texttt{"sel:bool"}1), and split the goal into two sub-goals to be attacked one at a time. The first is a rewrite of the original goal with \texttt{F}

\footnote{The HOL term for a case must type on its own, either manifestly, e.g. \texttt{"T"}, or explicitly, e.g. \texttt{"sel:bool"} or \texttt{"(f: num→bool) t: num"}.}
replacing every free occurrence of the argument. The other sub-goal is a
rewrite of the original sub-goal with T replacing every free occurrence of
the argument.

```
#e(BOOL_CASES_TAC "sel:bool");
OK.
2 subgoals
"(z = (\neg (a \land \neg F) \land (b \land F))) = (z = ((\neg F) \Rightarrow a \lor b))"
"(z = (\neg (a \land \neg T) \land (b \land T))) = (z = ((\neg T) \Rightarrow a \lor b))"
() : void
```

We now have two sub-goals to prove. HOL lets us attack them one at a
time; the textually lower one is the current goal. The goals can be simplified
considerably by applying simple rules involving and'ing and or'ing T and
F. Such rules are built into HOL as part of REWRITE_TAC. REWRITE_TAC is a
more powerful rewrite package than PURE_REWRITE_TAC. It not only rewrites
with the user supplied rewrite list, but also tries to apply several simple but
commonly required theorems which are built into HOL. PURE_REWRITE_TAC
is the faster of these two rewrite tactics and should be used in the early
stages of a proof elaboration when straightforward substitution of definitions
is all that is needed.

```
#e(REWRITE_TAC []);
OK.
goal proved
1- (z = (\neg (a \land \neg T) \land (b \land T))) = (z = ((\neg T) \Rightarrow a \lor b))

Previous subproof:
"(z = (\neg (a \land \neg F) \land (b \land F))) = (z = ((\neg F) \Rightarrow a \lor b))"
() : void
```

The rewrite was applied only to the current sub-goal which it was powerful
enough to solve. Clearly the same tactic will solve the second sub-goal too.

This is a good time to introduce how ML permits us to combine tactics
using the tactical THEN. THEN allows us to group several tactics as a single
syntactic unit. If t₁ and t₂ are tactics, then so is t₁ THEN t₂ and its effect
is to carry out t₁ and then to carry out t₂. If t₁ returns several sub-goals,
then t₂ is applied to each sub-goal.

We can restore the goal stack to its previous state with a call on backup
(or b for short) and this can be done anytime. b is very useful when we
wish to probe a little and experiment to find the smartest next tactic from
several possibilities (as here), when we realise that we have gone down a wrong line of attack and wish to recover, or if we spot a smarter manoeuvre and wish to tidy up a proof whilst it is still fresh in our mind.

To see the tactical THEN in action, we backup twice—first through the application of REWRITE_TAC and then through BOOL_CASES_TAC.

\[ \text{#b();;} \]
\[2\text{ subgoals} \]
\[
\begin{align*}
(z = \neg(\neg(a \land F) \land \neg(b \land P))) &= (z = (\neg F) \Rightarrow a \lor b)) \\
(z = \neg(\neg(a \land \neg T) \land \neg(b \land T))) &= (z = (\neg T) \Rightarrow a \lor b))
\end{align*}
\]
\[() : \text{ void} \]
\[\text{#b();;} \]
\[
\begin{align*}
(z = \neg(\neg(a \land \neg sel) \land \neg(b \land \neg sel))) &= (z = (\neg sel) \Rightarrow a \lor b))
\end{align*}
\]
\[() : \text{ void} \]

Then we apply the compound tactic BOOL_CASES_TAC "sel:bool" THEN REWRITE_TAC[]. As can be seen, this combined tactic has the desirable effect of not only splitting the goal into two legs but also simplifying (and in this case solving) both legs by use of REWRITE_TAC.

\[ \text{#e(BOOL\_CASES\_TAC "sel:bool" THEN REWRITE\_TAC []);;} \]
\[\text{OK..} \]
\[\text{goal proved} \]
\[
\begin{align*}
\vdash (z = \neg(\neg(a \land \neg sel) \land \neg(b \land sel))) &= (z = (\neg sel) \Rightarrow a \lor b)) \\
\vdash (\neg selbar p q. \\
(\neg selbar = \neg sel) \land \\
(p = \neg(a \land \neg selbar)) \land \\
(q = \neg(b \land selbar)) \land \\
(z = \neg(p \land q))) &= \\
(z = (\neg sel) \Rightarrow a \lor b)) \\
\vdash (\neg selbar p q. \\
\text{iev sel selbar /\ \negand2 a selbar p /\ \negand2 b sel q /\ \negand2 p q z) =} \\
(z = (\neg sel) \Rightarrow a \lor b)) \\
\vdash \text{mux2_imp sel a b z = mux2_spec sel a b z} \\
\vdash \text{a b out. mux2_imp sel a b z = mux2_spec sel a b z} \\
\vdash \text{sel a b out. mux2_imp sel a b z = mux2_spec sel a b z}
\end{align*}
\]
Previous subproof:
\[\text{goal proved} \]
\[() : \text{ void} \]

\[ \text{As we shall see, showing backup traces can waste a lot of space. In future, we will omit backup traces when they are non-trivial. After all, we have seen all the intermediate steps on the way.} \]
A surprising amount of material has been echoed back! When you do a tactical proof, each tactic has a corresponding forward inference rule that will generate the required theorem from the solved subgoals. When you exhaust one branch of the proof tree by solving all subgoals, the system applies the forward inference rules, generating the theorems that correspond to each subgoal in the proof, right up to the theorem that corresponds to the original goal. What you see echoed back are the actual theorems generated for each sub-goal in the proof to which a tactic was applied by use of the \texttt{expand} function. We would now like to save this theorem under its own name in our theory. This we do by using the built-in function \texttt{prove_thm}\[\begin{verbatim}
#prove_thm;
- : ((string # term # tactic) -> thm)
\end{verbatim}\]which requires a 3-tuple argument, a string (we will use \texttt{`mux2\_correct'}) which is the name under which the theorem will be stored in the theory, the goal, and the proof development, written here as the single tactic:

\begin{verbatim}
REPEAT GEN\_TAC
THEN PURE\_REWRITE\_TAC [ mux2\_imp; mux2\_spec; inv; nand2 ]
THEN EXISTS\_ELIM\_TAC
THEN BOOL\_CASES\_TAC "sel:bool"
THEN REWRITE\_TAC []
\end{verbatim}

The completed proof is given below bracketed between calls on the system routine \texttt{timer} which we use to gauge the complexity of a proof.

\begin{verbatim}
#timer true;;
false : bool
Run time: 0.0s

#let mux2\_correct = prove\_thm
 ("mux2\_correct",
 " ! sel a b z . mux2\_imp sel a b z = mux2\_spec sel a b z",
 REPEAT GEN\_TAC
 THEN PURE\_REWRITE\_TAC [ mux2\_imp; mux2\_spec; inv; nand2 ]
 THEN EXISTS\_ELIM\_TAC
 THEN BOOL\_CASES\_TAC "sel:bool"
 THEN REWRITE\_TAC []);
mux2\_correct = 1- !sel a b z . mux2\_imp sel a b z = mux2\_spec sel a b z
Run time: 4.6s
Intermediate theorems generated: 1169

#timer false;;
true : bool
\end{verbatim}

HOL responds by storing the theorem away in the current theory \texttt{gates} and echoing back what has been proved.
EXERCISES 7

Exercise 7.1 Specify the gates below, give implementation definitions, and correctness proofs.

\( \text{and}^2, \text{and}^3, \text{and}^4, \text{and}^5, \text{buffer}, \text{mux}^4, \text{or}^2, \text{or}^3, \text{or}^4, \text{or}^5, \text{xor}^2, \text{xnor}^2 \)

Except for the mux and the xor2 gates, all these gates may be constructed by taking an inverter onto an existing design. Extend the theory gates to contain all these definitions and proofs.

NB The tactic for mux4 and xor2 has the same form as that for mux2. The remaining proofs should all be a little simpler and be very similar in structure. You should be able to find a single (parameterised) tactic to take care of the and, buffer, or and xnor proofs.

Exercise 7.2 Here are the specifications of two circuits in HOL:


code21 = new_definition 
  ('encode21', 
   'encode21 a b z = (z = a => F | T)');

decode21 = new_definition 
  ('decode21', 
   'decode21 a z0 z1 = (z0 = a) \ (z1 = (~a) \ b)');

Define and verify implementations of these circuits. Specify and design encode42 and decode42 circuits.

Exercise 7.3 Collect together in a theory some useful facts about the associativity and distributivity of conjunction and disjunction. (CONJ_ASSOC is built-in.)

\[
\begin{align*}
\text{DISJ_ASSOC} &= |\neg | \ a \ b \ c \ . \ (a \lor (b \lor c)) = ((a \lor b) \lor c) \\
\text{DISJ_DISTRIBUT1} &= \neg \ (a \lor b \ c) \ . \ ((a \lor (b \lor c)) = ((a \lor b) \lor (a \lor c)) \\
\text{DISJ_DISTRIBUT2} &= \neg \ (a \lor b \ c) \ . \ ((b \lor c) \lor a) = ((b \lor a) \lor (c \lor a)) \\
\text{CONJ_ASSOC} &= \neg \ (a \lor b \ c) \ . \ (a \lor (b \lor c)) = ((a \lor b) \lor c) \\
\text{CONJ_DISTRIBUT1} &= \neg \ (a \lor b \ c) \ . \ ((a \lor (b \lor c)) = ((a \lor b) \lor (a \lor c)) \\
\text{CONJ_DISTRIBUT2} &= \neg \ (a \lor b \ c) \ . \ (b \lor (c \lor a)) = ((b \lor a) \lor (c \lor a))
\end{align*}
\]

You should be able to discover one simple tactic that will cope with all these cases.

If you need to apply any of these theorems “the other way” round, just use SYM_RULE, e.g.
#SYM_RULE;;  
- : (thm \rightarrow thm)

#DISJ_ASSOC;;  
1- !a b c. a \lor b \lor c = (a \lor b) \lor c

#SYM_RULE DISJ_ASSOC;;  
1- ! a b c. ((a \lor b) \lor c) = (a \lor (b \lor c))

**WARNING.** We assume the existence of these definitions and theorems from now on. They are proved the theory bools of appendix C.
Chapter 8

Bits, numbers, and words

When we are to carry out several proofs in one area of hardware, it pays to look for and collect any theorems that are of general use. We have just collected together a number of definitions and theorems about basic gates together in a theory. Now we look ahead and anticipate working with busses and number abstractions instead of boolean signals. In this chapter we gather together a number of definitions and theorems which will stand us in good stead when we go on to specify and verify combinational arithmetic circuits and subsystems.

- The theory `bits` contains definitions and theorems to map between booleans and numbers.
- The theory `nums` contains definitions and theorems concerning numbers.
- The theory `words` contains definitions and theorems to help us to abstract from bits on individual bus lines to numbers and to reason about their properties.

One section is devoted to each of these theories. At the head of each section we list the most useful theorems from the pertinent theory. In this chapter, we have chosen to work through a few carefully selected theorems, rather than detail all facets of these theories. Complete listings of the theories `bits`, `nums`, and `words` containing proofs of these and several other theorems is given in appendix C. We will assume the existence of these theories in future proofs, but leave it to you to acquaint yourself with the contents of the theories. It will pay you to work through the proofs in detail and further familiarise yourself with our style of proof.

Before presenting the major examples, we first present the techniques that are used in their solution.

8.1 Techniques I—arithmetic constants

Many hardware proofs involve arithmetic manipulations. In this section we show you how to simplify expressions involving arithmetic constants and the comparators $\neq, <, >, \leq$, and $\geq$. 
8.1.1 Conversion to standard form

The first step is to reduce the variety of required theorems by rewriting certain terms in goals to a standard form. For example, we convert all non-zero numbers into the successor notation (e.g. \(1 = \text{SUC}(0)\), \(2 = \text{SUC}(\text{SUC}(0))\) etc), using the conversion `num_CONV`. As a rider, we should write all our theorems in standard form (with \(\text{SUC}(0)\) for 1, \(\text{SUC}(\text{SUC}(0))\) for 2, etc) unless there is a good reason for not so doing.

```
num_CONV;;
- : conv

#( num_CONV "1", num_CONV "2" );;
(1-1 = SUC 0, 1-2 = SUC 1) : (thm # thm)

num_CONV "0";;
evaluation failed num_CONV: argument less than 1

#e "(2 = 0) \ (2 = 2) \ (1 = 2)";;
"(2 = 0) \ (2 = 2) \ (1 = 2)"

() : void

#e(PURE_REWRITE_TAC [ num_CONV "1"; num_CONV "2" ]);;
OK
"(\text{SUC}(\text{SUC}(0)) = 0) \ (\text{SUC}(\text{SUC}(0)) = \text{SUC}(\text{SUC}(0))) \ (\text{SUC}(0) = \text{SUC}(\text{SUC}(0)))"

() : void
```

A conversion takes an expression (say \(E\)) converts it to an equivalent expression (say \(E'\)) and returns a theorem \(\vdash E = E'\) which can then be used as a rewriting agent. Applying `CONV_RULE` to a conversion turns it into a rewrite rule; applying `CONV_TAC` to a conversion turns it into a tactic. Thus conversions are powerful primitives which, if used in this way, guarantee consistent treatments of rewriting.

8.1.2 Equality of arithmetic constants

With constants converted to SUC form, we turn our attention to dealing with conditions of the general form \(\text{SUC}^n(0) = \text{SUC}^m(0)\), where \(\text{SUC}^n\) means \(n\) repeated applications of \(\text{SUC}\).\(^1\) Amongst the built-in theorems of HOL we find the very useful

\(^1\)To be precise \(\text{SUC}^0 = 1\) and \(\text{SUC}^{n+1} = \text{SUC}^n \circ \text{SUC}.\)
Many of the built-in definitions and theorems such as INV_SUC_EQ must be retrieved from the appropriate theory. This is handled automatically, and announced by the autoloading message the first time the theorem is used in a session, as above. We shall not show such messages hereafter.

If we rewrite a term of the form $\textit{SUC}^n 0 = \textit{SUC}^m 0$ with INV_SUC_EQ, it will repeatedly strip away matching \textit{SUC}s from both sides of the equality until we wind up with one of (i) $0 = \textit{SUC}^k 0$, (ii) $0 = 0$, or (iii) $\textit{SUC}^k 0 = 0$, where $k > 0$. Two other theorems then come into play:

Applying \textit{SUC_NOT} will rewrite terms of the form (i) to F; \textit{REWRITE_TAC} will automatically rewrite terms of the form (ii) to T; and applying \textit{NOT_SUC} will rewrite terms of the form (iii) to F. We gather these rules together in a list:

We continue our little session showing these rewrite tactics in action. top_goal is a built-in function which echoes the current goal (useful if the latter has been scrolled off the screen due or does not appear on this page in the text!)
8.1.3 Less than comparisons

Comparisons with $<$ can be dealt with in a similar manner. Looking through the built-in theorems of HOL, we find

```
#let lssComp = [ LESS_MONO_EQ; LESS_0; LESS_REFL; NOT_LESS_0 ];
lessComp =
[ \m n. (SUC m) < (SUC n) = m < n;
  \m n. 0 < (SUC n);
  \m n. n < n;
  \m n. n < 0]
: thm list
```

Rewriting with LESS_MONO_EQ will convert any term of the form $SUC^n 0  < SUC^m 0$ to one of $0 < SUC^k 0$, $0 < 0$, or $SUC^k 0  < 0$, where $k > 0$. These will be simplified to T, F, or F respectively by rewriting with the built-in theorems LESS_0, LESS_REFL, or NOT_LESS_0.
8.1. TECHNIQUES I—ARITHMETIC CONSTANTS

\[ g \quad "(0 < 2) \land \neg (2 < 2) \land \neg (2 < 1)" \]

\[
(0 < 2) \land \neg 2 < 2 \land \neg 2 < 1
\]

( ) : void

\[ e \quad \text{REWRITE_TAC [ num\_CONV "1"; num\_CONV "2" ] } \]

OK.

\[ 0 < (SUC\_SUC\_0) \land \\
\neg (SUC\_SUC\_0) < (SUC\_SUC\_0) \land \\
\neg (SUC\_SUC\_0) < (SUC\_0) \land \\
\neg 0 < 2 \land \neg 2 < 2 \land \neg 2 < 1 \]

( ) : void

\[ e \quad \text{REWRITE_TAC less\_Comp} \]

OK.

goal proved

\[ |- \; 0 < (SUC\_SUC\_0) \land \\
\neg (SUC\_SUC\_0) < (SUC\_SUC\_0) \land \\
\neg (SUC\_SUC\_0) < (SUC\_0) \land \\
\neg 0 < 2 \land \neg 2 < 2 \land \neg 2 < 1 \]

Previous subproof:

goal proved

( ) : void

8.1.4 Other comparisons

Finally we note that the other comparators are definable in terms of = and <. When we don’t have ready-made theorems to hand, any term involving ≤, ≥, or > may be rewritten using one of the built-in theorems LESS OR EQ, GREATER OR EQ, or GREATER.

\[ \# \text{let op\_Comp = [ LESS\_OR\_EQ; GREATER\_OR\_EQ; GREATER ] } \]

op\_Comp =

\[ \text{[| \neg !m. n. m \leq n = m < n \lor (m = n); \\
\neg !m. n. m \geq n = m > n \lor (m = n); \\
\neg !m. n. m > n = n < m] : thm list} \]

This makes sense because most of the built-in HOL theorems on comparators prove facts about just = and <. Thus, given a goal like

\[ g \quad "(a < b) \land (c > d) \land (e \equiv f) \land (g \geq h)" \]

( ) : void

our strategy would be to put the goal in standard form by rewriting with op\_Comp
#e(PURE_REWRITE_TAC opComp);

OK.
"\(a < b \lor d < c \land (e < f \lor (e = f)) \lor (h < g \lor (g = h))\)"

() : void

## 8.2 Bits—useful facts about booleans

The theory `bits` contains a number of trivial yet useful theorems. As we saw in chapter 5, we get a very generic style for specifying arithmetic circuits if we look at values on the non-carry lines as numbers rather as booleans. `bv` is the primitive mapping which turns a boolean signal into a number. `bvals` converts boolean signals to numbers. `ivals`: `bvFF`, `bvFT`, and `bvTT`; `bvLss`, `bvEq`, and `bvGtr` simplify some common numeric expressions. `maxbit` and `maxbit2` enable us to reason about the maximum numeric values representable on wires (extended later to busses).

### Example 8.2.1

\(\forall a \ b . \ (bv \ a + bv \ b = 2) = a \land b\)

**Proof strategy.** The first steps are obvious: we remove the generalisations, and bool-case on `a` and `b`. When we rewrite, the right hand side of each case will simplify to `F` or `T`, and our four subgoals will be of the form `lhs = F` or `lhs = T`. These are rewritten at once to `~lhs` and `lhs` respectively. In either case we are left with numeric terms which we put in standard form using `bvals` and `num_CONV`. (`bvals` converting `bv` to `0` and `bv` to `SUC 0`.)
8.2. BITS—USEFUL FACTS ABOUT BOOLEAN

When we bool case on a and b, instead of writing

```
BOOL_CASES_TAC "a:bool" THEN BOOL_CASES_TAC "b:bool"
```

we write

```
MAP_EVERY BOOL_CASES_TAC [ "a:bool" ; "b:bool" ]
```

This tactic first (this is where MAP comes in) constructs the list of tactics

```
[ BOOL_CASES_TAC "a:bool" ; BOOL_CASES_TAC "b:bool" ]
```

and then (this is where EVERY comes in) applies the tactics in the list one by one. MAP, EVERY, and MAP_EVERY are detailed in part V.

**Proof.** We assume that bv and bvals are to hand. The first steps need no comment.

```ml
#e (REPEAT GEN_TAC
    THEN MAP_EVERY BOOL_CASES_TAC [ "a:bool" ; "b:bool" ]
    THEN REWRITE_TAC [ bvals ; num_CONV "1" ; num_CONV "2" ]);
OK.
4 subgoals

"(0 + 0 = SUC(SUC 0))"

"(0 + (SUC 0) = SUC(SUC 0))"

"((SUC 0) + 0 = SUC(SUC 0))"

"(SUC 0) + (SUC 0) = SUC(SUC 0)"

() : void
```

The built-in theorem **ADD_CLAUSES** will remove any additions of zero (clauses 1 and 2). Clause 4 will also rewrite the term SUC 0 + SUC 0 to SUC(SUC 0). This puts all 4 subgoals into forms that can be dealt with by rewriting with the theorems in the list **eq1Comp**. We backup and show how it is done. Notice how we join the two rewrite lists together with @. They are, after all, ML lists.
8.3 Techniques II—the assumption list

Suppose we are faced with a theorem which has the form of an implication, for example.

```plaintext
#g "(a = b) \implies \neg (a < b) /\ \neg (a > b)";;
"(a = b) \implies \neg a < b /\ \neg a > b"
```

() : void

```plaintext
#e(PURE_ONCE_REWRITE_TAC [ GREATER ]);;
OK .
"(a = b) \implies \neg a < b /\ \neg b < a"
```

() : void
8.3. **Techniques II—The Assumption List**

We first put the goal in standard form by rewriting with **GREATER**. By applying **DISCH_TAC** we then move the antecedent onto the assumption list where it can be used directly as a rewriting agent. **DISCH_TAC** takes a goal of the form \((G, \ a \implies \ b)\) to a goal of the form \((G + \ a, \ b)\).

```ml
#e(DISCH_TAC);;
OK.
"" a < b /\ \ " b < a"
  [ "a = b" ]
()
```

Rewriting with **LESS_REFL** solves this goal.

```ml
#LESS_REFL;;
|\- \!n. \ a < n

#e(REWRITE_TAC [ LESS_REFL ]);;
OK.
goal proved
|\- \ a < b
  . |\- \ a < b /\ \ ~ b < a
|\- (a = b) \implies \ a < b /\ \ ~ b < a
|\- (a = b) \implies \ ~ a < b /\ \ ~ a > b

Previous subproof:
goal proved
()
```

Sometimes the antecedent cannot be used directly and we have more work to do. Consider the following (rather contrived) example:

```ml
#e " ! m n . (SUC m < n) \implies \ "(m = n)"; 
"! m n . (SUC m) < n \implies \ "(m = n)"
()
```

Rewriting with **REPEAT GEN_TAC THEN DISCH_TAC**;

```ml
OK.
""(m = n)"
  [ "(SUC m) < n" ]
()
```

CHAPTER 8. BITS, NUMBERS, AND WORDS

Nothing will be gained by attempting a rewrite with this assumption. However amongst the built-in theorems of HOL we find

\begin{verbatim}
#(SUC_LESS, LESS_NOT_EQ);
(\|- !m n. (SUC m) < n ==> m < n, \|- !m n. m < n ==> ~(m = n))
: (thm # thm)
\end{verbatim}

We now introduce the \texttt{thm tac} \texttt{IMP_RES_TAC}. A \texttt{thm tac} takes a theorem and generates a tactic which it then applies to the current goal. \texttt{IMP_RES_TAC} takes a theorem, say \texttt{th1}, in the form of an implication and tests it against each and every assumption. Every time an assumption can be matched against the antecedent of \texttt{th1}, an appropriately instantiated consequent is added to the assumption list. The goal is solved in just two steps. The second application of \texttt{IMP_RES_TAC} puts a copy of the goal on the assumption list and this is sufficient to solve the goal.

\begin{verbatim}
#(IMP_RES_TAC SUC_LESS);
OK.
"(m = n)"
[ "!(SUC m) < n" ]
[ "m < n" ]
()
: void

#(IMP_RES_TAC LESS_NOT_EQ);
OK.
goal proved
. \|- ~(m = n)
. \|- ~(m = n)
\|- !m n. (SUC m) < n ==> ~(m = n)

Previous subproof:
goal proved
()
: void
\end{verbatim}

8.4 Nums—useful facts about numbers

The theory \texttt{nums}

\begin{verbatim}
multby2     = \|- !n. 2 * n = n + n
mult_eq_0   = \|- !a b. (a * b = 0) = (a = 0) \lor (b = 0)
less_less_add = \|- !a b x. x < a ==> x < (a + b)
neg_add_less = \|- !a b. ~(a + b) < b
less_less_add_iss = \|- !a b c d. a < b \land c < d ==> (a + c) < (b + d)
\end{verbatim}
The theory \texttt{nums} gathers together a few theorems that we have found useful when verifying arithmetic circuits. The first two theorems are obvious. The next three are used frequently at the leaf level of arithmetic proofs. The next four fill in a few holes in the HOL collection of theorems about subtraction. The final theorem gathers together a number of trivial facts about subtraction and parallels the built-in \texttt{ADDCLAUSES} and \texttt{MULTCLAUSES}.

**Example 8.4.2** \(\forall a b x. (x < a) \implies x < (a + b)\)

In this proof we find that the antecedent cannot be used directly when we push it onto the assumption list.

**Proof strategy.** We tackle this theorem by cases. If \(b \neq 0\), the proof is trivial. When \(b = 0\), we use the built-in theorem

\begin{verbatim}
#LESS_ADD_NONZERO;;
|- !m n. (n = 0) ==> m < (m + n)
\end{verbatim}

Given that \(b \neq 0\), we can infer from \texttt{LESS\_ADD\_NONZERO} that \(a < a + b\). Given that \(x < a\) and that \(a < a + b\), we use the built-in theorem

\begin{verbatim}
#LESS\_TRANS;;
|- !m n p. m < n \&\& n < p ==> m < p
\end{verbatim}

To infer that \(x < a + b\).

**Proof.** We start by setting the goal and removing the generalised variables.

\begin{verbatim}
# " ! a b x. (x < a) ==> x < (a + b)";;
"! a b x. x < a ==> x < (a + b)"

0: void
\end{verbatim}

We now case split on b = 0. If we use `BOOL_CASES_TAC` then we lose knowledge of which case we are dealing with. Here it is crucial that this information be retained. In situations like this, we use `ASM_CASES_TAC` which puts the pertinent case explicitly on the assumption list where it can be used for rewriting now and where it remains available for later use (drawing extra inferences via `IMP_RES_TAC`).

If we rewrite the top goal from the assumption list, the term on the right in the top goal becomes a + 0. Further rewriting with `ADD_CLAUSES` is all we need.

The next stage of the proof is to get some additional helpful facts onto the assumption list. One of these is \( x < a \).

To get \( a < (a + b) \) onto the assumption list, we try applying `IMP_RES_TAC` to the built-in theorem `LESS_ADD_NONZERO`.
### S.4. Nums—Useful Facts About Numbers

<table>
<thead>
<tr>
<th>LESS_ADD_NONZERO;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>OK.</td>
</tr>
<tr>
<td>&quot;x &lt; (a + b)&quot;</td>
</tr>
<tr>
<td>[ &quot;&quot; (b = 0)&quot; ]</td>
</tr>
<tr>
<td>[ &quot;x &lt; a&quot; ]</td>
</tr>
<tr>
<td>[ &quot;!m. m &lt; (m + b)&quot; ]</td>
</tr>
<tr>
<td>() : void</td>
</tr>
</tbody>
</table>

But this does not appear to be quite what we want. **IMP_RES_TAC** generalizes every variable which was not fixed by the match with the antecedent part of the theorem. Here the assumption (b = 0) was matched to (n = 0), so the variable m remains quantified in the added assumption. Notice that specializing this variable to say a will not prevent its being generalized. We try **IMP_RES_TAC** again, this time with **LESS_TRANS** as argument.

<table>
<thead>
<tr>
<th>LESS_TRANS;</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK.</td>
</tr>
<tr>
<td>&quot;x &lt; (a + b)&quot;</td>
</tr>
<tr>
<td>[ &quot;&quot; (b = 0)&quot; ]</td>
</tr>
<tr>
<td>[ &quot;x &lt; a&quot; ]</td>
</tr>
<tr>
<td>[ &quot;!m. m &lt; (m + b)&quot; ]</td>
</tr>
<tr>
<td>[ &quot;!p. a &lt; p =&gt; x &lt; p&quot; ]</td>
</tr>
<tr>
<td>[ &quot;!m. m &lt; x =&gt; m &lt; a&quot; ]</td>
</tr>
<tr>
<td>() : void</td>
</tr>
</tbody>
</table>

**IMP_RES_TAC** is not smart enough to pick up and conjoin the assumptions x < a and !m. m < m + b, specialize m to a and from them infer that x < a + b. Without specializing m resolution will not produce the result we want, so we back up and try a different approach, this time using **MATCH_MP_TAC** to match the consequent of **LESS_TRANS** with the goal, rather than matching assumptions.
This time the variable $n$ is not fixed by the match and is therefore existentially quantified in the new subgoal. The value we want is $a$, and then rewriting with the assumptions will solve the goal.

We save the theorem under the name $\text{less\_less\_add}$.
8.5 Techniques III—induction

We use the simple—yet oft-used—built-in theorem LESS_MONO_ADD_EQ to illustrate the mechanics of proofs by induction.

```
#let lss_lss_add = prove_thm
   (`lss_lss_add`,
    "! a b x. (x < a) ==> x < (a + b)",
    REPEAT GEN_TAC THEN STRIP_TAC
    THEN ASM_CASES_TAC "b = 0"
    THENL
    [ ASM_REWRITE_TAC [ ADD_CLAUSES ]
      ; IMP_REWRITE_TAC LESS_ADD_NONZERO
      THEN MATCH_MP_TAC LESS_TRANS
      THEN EXISTS_TAC "a:num"
      THEN ASM_REWRITE_TAC []
    ]
    lss_lss_add = |- !a b x. x < a ==> x < (a + b)
```

8.5 Techniques III—induction

We use the simple—yet oft-used—built-in theorem LESS_MONO_ADD_EQ to illustrate the mechanics of proofs by induction.

```
#g " ! m n p. (m + p) < (n + p) = m < n";;
"! m n p. (m + p) < (n + p) = m < n"
() : void
```

We first select the generalised variable on which we wish to induct. In this case, we choose to induct on \( p \). The fact that it appears symmetrically on the left hand side gives us very desirable properties when rewriting. We start by picking our way carefully past \( m \) and \( n \).

```
#e(GEN_TAC THEN GEN_TAC THEN INDUCT_TAC);;
OK.
2 subgoals
"(m + (SUC p)) < (n + (SUC p)) = m < n"
[ "(m + p) < (n + p) = m < n" ]
"(m + 0) < (n + 0) = m < n"
() : void
```

INDUCT_TAC splits the goal "! p . (m + p) < (n + p) = m < n" into two subgoals; a base case (the top subgoal) and the inductive case. Notice how the induction assumption is placed on the assumption list for the inductive case. Notice also the “nice” properties of both subgoals due to our inducting on \( p \).

The base case is easy to solve: we just rewrite with ADD_CLAUSES
Once the base case has been solved, the inductive step is automatically popped up. Here the inductive case is also easy. We remind you of \texttt{ADD\_CLAUSES}:

\begin{verbatim}
#ADD\_CLAUSES;
  "(0 + m = m) /
  "(m + 0 = m) /
  "((SUC m) + n = SUC(m + n)) /
  "(m + (SUC n) = SUC(m + n))
\end{verbatim}

Clause 4 rewrites the left hand side into a form where both \texttt{SUC}s can be cancelled.

\begin{verbatim}
#e(REWRITE\_TAC [ ADD\_CLAUSES ]);
OK.
"
"(SUC(m + p)) < (SUC(n + p)) = m < n"
  [ "(m + p) < (n + p) = m < n" ]
() : void
#e(REWRITE\_TAC [ LESS\_MONO\_EQ ]);
OK.
"
"(m + p) < (n + p) = m < n"
  [ "(m + p) < (n + p) = m < n" ]
() : void
\end{verbatim}

Now the goal is an exact copy of the induction assumption and all that is left is a simple rewrite from the assumption list.
8.6. Words—Values on Busses

Here is the tidy form of the proof:

```ml
#let LESS_MONO_EQ = prove_thm
  ('LESS_MONO_ADD_EQ',
   " ! m n p . (m + p) < (n + p) = m < n",
   GEN_TAC THEN GEN_TAC THEN INDUCT_TAC
   THEN ASM_REWRITE_TAC [ADD_CLAUSES; LESS_MONO_EQ]);;
LESS_MONO_EQ = |- !m n p. (m + p) < (n + p) = m < n
```

The strategy behind inductive proofs is as follows. Where there is a choice think carefully about which variable (or variables) to induct on. If in doubt, experiment and backup. Two cases will appear for each application of INDUCT_TAC. The base case should be trivial. In inductive step the basic idea is to reshape the goal until the induction hypothesis can be applied by rewriting from the assumption list.

8.6 Words—Values on Busses

The theory words

```
exp_pos      = |- !n. 0 < (2 EXP n)
exp_not_0    = |- !n. '-(2 EXP n = 0)
exp_mono     = |- !n. (2 EXP n) < (2 EXP (SUC n))
exp_doubles  = |- !n. 2 EXP (SUC n) = (2 EXP n) + (2 EXP n)

val          = |
  |- (!f. val f 0 = bv(f 0)) /
               (!n f. val f(SUC n) = val f n + ((2 EXP (SUC n)) * (bv(f(SUC n)))))

maxword      = |- !n a. (val a n) < (2 EXP (SUC n))
maxword2     = |- !n a cin. ((val a n) + (bv cin)) <= (2 EXP (SUC n))
```

The theory contains several basic theorems about the exponential function. `val` computes the num value on a buss from its bits. `maxword` and `maxword2` tell us facts about the maximum values of bits on busses.
When comparing bit values we used the abstraction $bv$. Likewise when we compare values on $n$-bit busses we use the abstraction $val$ discussed in chapter 5. In HOL, we define primitive recursive functions using

\begin{verbatim}
#new_prim_rec_definition;;
- : ((string # term) -> thm)
\end{verbatim}

$new\_prim\_rec\_definition$ is a built-in function which takes a pair—a string (save name) and term—as its argument. The term must be the conjunction of a base case definition and an inductive step definition of the function. For example here is the HOL definition of the built-in infix operator $\text{EXP}$.

\begin{verbatim}
#EXP;;
l- (!m. m EXP 0 = 1) /
     (!m n. m EXP (SUC n) = m * (m EXP n))
\end{verbatim}

and here is the definition of $val$ in HOL.

\begin{verbatim}
#let val = new_prim_rec_definition
  ('val',
   " (val (f:num->bool) 0 = (bv(f 0)))
     /
     (val f (SUC n) = (val f n) + ((2 EXP (SUC n)) * bv(f(SUC n))))
   "));;
val =
l- (!f. val f 0 = bv(f 0)) /
     (!f n. val f (SUC n) = (val f n) + ((2 EXP (SUC n)) * (bv(f(SUC n)))))
\end{verbatim}

Just as with non-recursive definitions, free variables in primitive recursive definitions are automatically generalised.

**Example 8.6.3** $\vdash \forall n\ a\ .\ val\ a\ n < (2 \ EXP \ (SUC\ n))$

**Proof strategy.** The proof is a simple induction on the bus width $n$. The goal is rewritten to a form involving an exponential, and we show one of the standard ways of dealing with these terms.

**Proof.** The proof starts with an induction on $n$ and our rewriting with $val$, but not with $\text{EXP}$ for a reason that becomes clearer as we go.
#g "! n a. (val a n) < (2 EXP (SUC n))";;
"!n a. (val a n) < (2 EXP (SUC n))"

() : void

#e (INDUCT_TAC THEN GEN_TAC
  THEN PURE_ONCE_REWRITE_TAC [ val ]));;
OK.
2 subgoals
"((val a n) + ((2 EXP (SUC n)) * (bv(a(SUC n)))) < (2 EXP (SUC(SUC n))))" 
  [ "!a. (val a n) < (2 EXP (SUC n))"
     ]
"((bv(a 0)) < (2 EXP (SUC 0))"

() : void

Base case. As is usual the base case presents no problems. Rewriting with
(the second conjunct in) EXP

#EXP;;
|- (!m. m EXP 0 = 1) /
    (!m n. m EXP (SUC n) = n * (m EXP n))

turns the right hand side of the goal into 2 * (2 EXP 0). Rewriting a
second time, this time with the first conjunct of EXP, simplifies the sub-goal
to 2 * 1. Rewriting with multby2 turns this into 2 and then the goal is
an exact copy of the conclusion of maxbit.

#e (PURE_REWRITE_TAC [ EXP; MULT_CLAUSES; maxbit ]);;
OK.
goal proved
|- (bv(a 0)) < (2 EXP (SUC 0))

Previous subproof:
"((val a n) + ((2 EXP (SUC n)) * (bv(a(SUC n)))) < (2 EXP (SUC(SUC n))))" 
  [ "!a. (val a n) < (2 EXP (SUC n))"
     ]

() : void

Inductive step. The inductive step contains an exponential term multi-
plied by a term which reduces to either zero or a one. The easiest way to
proceed is to bool-case on a(SUC n), then to use bvals to turn bv F to 0
and bv T to 1. MULT_CLAUSES simplifies both products, and the resulting
zero term (case b(SUC n) = F) is removed by rewriting with ADD_CLAUSES.
Two subcases result, neither of which is hard. The top goal has the form \( a + b < b^2 \) which we rewrite to the form \( a + b < b + b \) using \texttt{exp doubles}. We can then cancel the \( b \) term from both sides.

Unfortunately rewriting with \texttt{exp doubles} does too much. (It is for this reason that we did not rewrite with \texttt{EXP} right at the beginning). We wish to simplify the term on the right only. Accordingly we back up and specialise \texttt{exp doubles} into a form with two occurrences of \texttt{SUC} so that it pattern matches only on the right.
8.6. WORDS—VALUES ON BUSSES

```
#b();
2 subgoals
"(val a n) < (2 EXP (SUC(SUC n)))"
[ "(a. (val a n) < (2 EXP (SUC n)))"
"((val a n) + (2 EXP (SUC n))) < (2 EXP (SUC(SUC n)))"
[ "(a. (val a n) < (2 EXP (SUC n)))"
]
()

#SPEC "SUC n" exp_double;;
|- 2 EXP (SUC(SUC n)) = (2 EXP (SUC n)) + (2 EXP (SUC n))
#e(PURE_ONCE_REWRITE_TAC [ (SPEC "SUC n" exp_double) ]);;
OK.
"((val a n) + (2 EXP (SUC n))) < ((2 EXP (SUC n)) + (2 EXP (SUC n)))"
[ "(a. (val a n) < (2 EXP (SUC n)))"
]
()
```

We now need a way to cancel the exponential term from both sides of the goal. We have already proved the appropriate theorem:

```
#LESS_MONO_ADD_EQ;;
|- m n p. (m + p) < (n + p) = m < n
#e(REWRITE_TAC [ LESS_MONO_ADD_EQ ]);;
OK.
"(val a n) < (2 EXP (SUC n))"
[ "a. (val a n) < (2 EXP (SUC n))"
]
()
```

```
#e(ASM_REWRITE_TAC []);
OK.
goal proved
.- (val a n) < (2 EXP (SUC n))
.- ((val a n) + (2 EXP (SUC n))) < ((2 EXP (SUC n)) + (2 EXP (SUC n)))
.- ((val a n) + (2 EXP (SUC n))) < (2 EXP (SUC(SUC n)))

Previous subproof:
"(val a n) < (2 EXP (SUC(SUC n)))"
[ "a. (val a n) < (2 EXP (SUC n))"
]
()
```

The last part of the proof uses modus ponens and the built-in theorem LESS_TRANS.
If we can fashion theorems $\text{lem}_1 = \vdash \text{val } a \ n < 2 \ \text{exp} \ (\text{suc } n)$ and $\text{lem}_2 = \vdash 2 \ \text{exp} \ (\text{suc } n) < 2 \ \text{exp} \ (\text{suc } (\text{suc } n))$, then $\text{mp} \ \text{less_trans} (\text{lem}_1 \ \land \ \text{lem}_2)$ will prove $\text{val } a \ n < 2 \ \text{exp} \ (\text{suc } (\text{suc } n))$.

$\text{lem}_1$ is a specialisation of the assumption. A straightforward way of gaining access to a term on the assumption list is to assume it. $\text{assume } t$ returns the theorem $t \vdash t$. To get the precise theorem we require, we specialise this theorem as shown

```
#let lem1 = spec_all (assume "\[ a . (val a n) < (2 \ \text{exp} \ (\text{suc } n))\]");
lem1 = \[ \vdash (val a n) < (2 \ \text{exp} \ (\text{suc } n))\]
```

Why not assume $\text{val } a \ n < 2 \ \text{exp} \ (\text{suc } n)$? The point is that we want to use $\text{lem}_1$ as a rewriting agent and when we do the HOL checks to see that no new assumptions are generated. The assumption of $\text{lem}_1$ is already on the assumption list; that of $\text{assume } \text{\"(val a n) < (2 \ exp (suc n)\)\"}$ is not.

$\text{lem}_2$ is a specialisation of $\text{exp\_mono}$.

```
#let lem2 = spec "suc n" exp_mono;
lem2 = \[ \vdash (2 \ \text{exp} \ (\text{suc } n)) < (2 \ \text{exp} \ (\text{suc } (\text{suc } n)))\]
```

These two theorems are conjoined together using $\text{conj}$. Doing $\text{modus ponens}$ on $\text{less_trans}$ gives us the theorem we need.

```
#conj lem1 lem2;
\[ \vdash (val a n) < (2 \ \text{exp} \ (\text{suc } n)) \land \ (2 \ \text{exp} \ (\text{suc } n)) < (2 \ \text{exp} \ (\text{suc } (\text{suc } n)))\]
```

We could rewrite with this theorem, but there is a faster way since it precisely matches the goal, and that is to apply $\text{accept\_tac}$ to it. $\text{accept\_tac}$ is a $\text{thm\_tactic}$ which matches its argument against the current goal. If they are identical, it solves the goal. Otherwise, it fails.
Here is the proof in tidy form:

```plaintext
#let maxword = prove_thm
   ('maxword',
    "! n a . (val a n) < (2 EXP (SUC n))",
    INDUCT_TAC THEN GEN_TAC
    THEN PURE_ONCE_REWRITE_TAC [ val ]
    THENL
    [ PURE_REWRITE_TAC [ EXP; MULT_CLAUSES; maxbit ]
      ; BOOL_CASES_TAC "(a(SUC n)):bool"
      THEN REWRITE_TAC [ bvals; ADD_CLAUSES; MULT_CLAUSES ]
      THENL
      [ ASM_REWRITE_TAC
        [ SPECF "SUC n" exp_doubles; LESS_MONO_ADD_EQ ]
        ; let lem1 = SPEC_ALL
          (ASSUME "!n . (val a n) < (2 EXP (SUC n))") in
        let lem2 = SPEC "SUC n" exp_mono in
        ACCEPT_TAC (MATCH_MP LESS_TRANS (CONJ lem1 lem2))
      ]
    ]
);;
maxword = |- !n a . (val a n) < (2 EXP (SUC n))
```
EXERCISES 8

Exercise 8.1 Prove the following theorems from nums.

\[
\text{maxbit2} = \begin{cases} \text{b} & \text{b} \leq 1 \\ \text{mult}_\text{eq}_0 & \text{a b} \cdot (a \cdot b = 0) = (a = 0) \land (b = 0) \\
\end{cases}
\]

Exercise 8.2 Prove the following theorems (all of which are built-in).

\[
\begin{align*}
\text{ADD_SYM} &= \begin{cases} \text{m n} & m + n = n + m \\
\text{LESS_TRANS} &= \begin{cases} \text{m p} & m < n \land n < p \Rightarrow m < p \\
\text{LESS_MONO_EQ} &= \begin{cases} \text{m} & (\text{SUC m}) < (\text{SUC n}) = m < n \\
\text{INV_SUC_EQ} &= \begin{cases} \text{m} & (\text{SUC m} = \text{SUC n}) = (m = n) \\
\text{EQ_MONO_ADD_EQ} &= \begin{cases} \text{m p} & (m + p = n + p) = (m = n)
\end{cases}
\end{cases}
\end{cases}
\end{align*}
\]

Exercise 8.3 Prove nLssVal = \begin{cases} \text{a b n} & (\text{lss a b n}) = (\text{val a n} < \text{val b n}) \\
\end{cases}

Exercise 8.4 Prove nEqVal = \begin{cases} \text{a b n} & (\text{eql a b n}) = (\text{val a n} = \text{val b n}) \\
\end{cases}

Exercise 8.5 Prove nGtrVal = \begin{cases} \text{a b n} & (\text{gtr a b n}) = (\text{val a n} > \text{val b n}) \\
\end{cases}

Exercise 8.6 Rework the proof of the word comparator by substituting in the definition of nComp_spec for the val expressions in terms of lss, eql, and gtr (use SYN_RULE and the three theorems above). The proof is rather simpler because these abstractions model the way the implementation is wired, and all the hard work involving arithmetic and exponents has already been done.
Chapter 9
Comparators

This chapter further explores the specification and verification of combinational devices. It stresses the importance of exploring facts about specifications, shows you how to deal with hierarchy, how to construct theorems on the fly, and how to use primitive induction tactics to tackle the verification of regular sub-systems.

We use the comparator as a running example. We first specify and verify a simple 1-bit comparator, extend it, and then build a word comparator by wiring several extended 1-bit comparators together.

### 9.1 A simple 1-bit comparator.

![Figure 9.1 Comp](image)

Comp takes in two 1-bit signals \(a\) and \(b\) and outputs three 1-bit signals \(l\), \(e\), and \(g\) which tell us whether the value on \(a\) is less, equal, or greater (respectively) than the value on \(b\).

#### 9.1.1 Specification

The specification of this device is straightforward. We write down what we expect to see on each of the three output lines and conjoin them on the right hand side. One way of writing the specification “thinks in terms of boolean values” and reads

\[
\text{comp } a \ b \ l \ e \ g \\
= (l = (\neg a \land b)) \land (e = (a = b)) \land (g = (a \land \neg b))
\]
but since comparison is a numeric operation we prefer the variant below which “thinks in terms of numeric values”

\[
\text{comp } a \ b \ l \ e \ g = (l = (b \ a < b \ b)) \land (e = (b \ a = b \ b)) \land (g = (b \ a > b \ b)).
\]

We will find that the latter style of specification extends to the word comparator in a straightforward fashion.

### 9.1.2 Consequences of the specification

Before moving on to defining an implementation we take time out to explore this specification. One obvious consequence should be that one (and only one) output line may be high. Does our specification pass this test? Let us see using HOL.

We start a fresh session by logging into HOL and opening a new theory \texttt{comp}. We bring down our libraries of definitions and facts about bools, bits, nums, words and gates and then enter the specification of \texttt{comp}. 

```plaintext
#new theory 'comparators';;
() : void

#let getTheories L
  = ( map new_parent L;
     map load_definitions L;
     map load_theorems L;
   );;
getTheories = - : (string list -> void list list)

#getTheories [ 'bools'; 'bits'; 'nums'; 'words'; 'gates' ];;
Theory bools loaded
Theory bits loaded
Theory nums loaded
Theory words loaded
Theory gates loaded
% << ***** listing omitted ***** >> %
```
9.1. A SIMPLE 1–BIT COMPARATOR.

We now make two auxiliary definitions which make precise what we mean by “one and only one line value is high”. `oneHigh a b c` says that if `a` is high then `b` is low and `c` is low. `oneAsserted a b c` states that at least one of `a` and `b` and `c` is high, and then ensures that if any one of these three values is high, the other two are not.

We now set our goal—over all inputs and outputs, the comparator specification implies that one and only one output is high. The first steps in the proof are obvious: remove the quantifications and rewrite the goal, and then move the antecedent to the assumption list.
CHAPTER 9. COMPARATORS

```ml
#e(REPEAT GEN_TAC
   THEN PURE_REWRITE_TAC
   [ Comp_spec; on_asserted; on_olhig; GREATER ]);;
OK . .
"(1 = (bv a) < (bv b)) \land (e = (bv a = bv b)) \land (g = (bv b) < (bv a)) \implies
 (1 \not< e \\not< g) \land
 ((1 \equiv' e) \land (1 \equiv' g)) \land
 ((e \equiv' g) \land (e \equiv' 1)) \land
 (g \equiv' 1) \land
 (g \equiv' e)"
() : void
```

If we were now to use DISCH_TAC, the antecedent would be placed on the assumption list in the form of a single conjunct. Instead we apply the more powerful tactic STRIP_TAC which breaks conjoined terms apart and, rather more conveniently, places each one on the assumption list separately. ¹ We at once rewrite with these assumptions.

```ml
#e(STRIPTAC
   THEN PURE_ASM_REWRITE_TAC []);
OK . .
"((bv a) < (bv b) \lor (bv a = bv b) \lor (bv b) < (bv a)) \land
 (((bv a) < (bv b) \implies (bv a = bv b)) \land
 ((bv a) < (bv b) \implies (bv b) < (bv a))) \land
 (((bv a = bv b) \implies (bv b) < (bv a)) \land
 (((bv a = bv b) \implies (bv a) < (bv b)) \land
 ((bv b) < (bv a) \implies (bv a) < (bv b)) \land
 ((bv b) < (bv a) \implies (bv a = bv b))")
 [ "1 = (bv a) < (bv b)" ]
 [ "e = (bv a = bv b)" ]
 [ "g = (bv b) < (bv a)" ]
() : void
```

Although the new goal is long it is quite simple in form. Rewriting with

```ml
#( LESS_NOT_EQ, NOT_LESS_EQ );
(\prefix \neg m \ n \ m < n \implies (m = n), \prefix \neg m \ n \ (m = n) \implies \neg m < n) : (thm \# thm)
```

will remove occurrences of

```
bv a < bv b \implies \neg (bv a = bv b)
bv a = bv b \implies \neg (bv a < bv b)
```

and rewriting with the equalities rotated

¹STRIP_TAC will also strip away quantified variables and case split on an antecedent which is a number of disjunctions.
will remove 2 more cases. Finally the theorem

will remove 2 more cases. Finally the theorem

which is proved in the theory nums will remove two more cases. We now rewrite with the above five theorems.

There is no built-in theorem to solve the above goal. We have to fashion a theorem of our own. Fortunately, this can be done on the fly as we are allowed to set another goal on top of the goal stack at any time. When we have proved the auxiliary theorem, we clear any clutter left over from it on the goal stack by backing-up, and then we are automatically returned to the main proof in the state in which we left it.

The goal is a special case of

\[ \forall a \ b \ c \ . \ (a < b) \lor (a = b) \lor (a > b) \]

and we prove the general case. We start by setting the goal, rewriting with GREATER and then inducting on both generalised variables.
Although we have four cases, they are all trivial, and we have seen how to deal with most of these subgoals. Notice how the induction assumptions appear. In particular that the bottom goal has two sets of assumptions to work with. We backup and rewrite.

The last subgoal is clearly solved by rewriting from the assumptions.
9.2. TECHNIQUES V — THEOREMS ON THE FLY

Here is the theorem in tidy form:

```ml
#let LorEorG = prove
  ("! a b . (a < b) \lor (a = b) \lor (a > b)",
   PURE_REWRITE_TAC [ GREATER ]
   THEN REPEAT INDUCT_TAC
   THEN ASM_REWRITE_TAC
   [ NOT_LESS_0; LESS_0; LESS_MONO_EQ; INV_SUC_EQ ];;LorEorG = !a b. a < b \lor (a = b) \lor a > b
```

... and back to the main theorem

We now remove the clutter on top of the goal stack left over from the proof of LorEorG and return to the main proof.

```ml
#b();;
"a < b \lor (a = b) \lor b < a"
[ "!b. a < b \lor (a = b) \lor b < a" ]
[ "(SUC a) < b \lor (SUC a = b) \lor b < (SUC a)" ]
() : void
#b();;
"!a b. a < b \lor (a = b) \lor a > b"
() : void
#b();;
"(bv a) < (bv b) \lor (bv a = bv b) \lor (bv b) < (bv a)"
[ "l = (bv a) < (bv b)" ]
[ "e = (bv a = bv b)" ]
[ "g = (bv b) < (bv a)" ]
() : void
```

The rewriting agent we require is a specialisation of the LorEorG

```ml
#PURE_ONCE_REWRITE_RULE [ GREATER ] LorEorG;;
!a b. a < b \lor (a = b) \lor b < a
```
which can be pattern matched to the goal. Accordingly we solve the goal
by applying MATCH ACCEPT_TAC, which is a pattern-matching generalisation
of ACCEPT_TAC

```ml
# MATCH_ACCEPT_TAC;;
- : thm_tactic

# (MATCH_ACCEPT_TAC (PURE_ONCE_REWRITE_RULE [ GREATER ] LorEorG));;
ok

goal proved

% << ***** trace omitted ***** >> %

|- !a b l e g. Comp_spec a b l e g ==> oneAsserted l e g

Previous subproof:
  goal proved
  () : void

```

Here is the proof in tidy form:

```ml
#let lem1 = prove
  (! !a b l e g
   Comp_spec a b l e g ==> oneAsserted l e g)

  THEN PURE_REWRITE_TAC [ Comp_spec; oneAsserted; oneHigh; GREATER ]

  THEN STRIP_TAC

  THEN ASM_REWRITE_TAC
    [ LESS_NOT_EQ; GSYM LESS_NOT_EQ;
    NOT_LESS_EQ; GSYM NOT_LESS_EQ;
    less_not_rev;
    ]

  THEN MATCH_ACCEPT_TAC
    (PURE_ONCE_REWRITE_RULE [ GREATER ] LorEorG)
);

lem1 = |- !a b l e g. Comp_spec a b l e g ==> oneAsserted l e g
```

Proving this fact gives us more confidence in our specification and we now
turn our attention to defining an implementation.

### 9.2.1 Implementation

For many simple circuits, a reasonable implementation can be derived from
the specification by calculation. The specification of \texttt{comp} states that the
value on the output line \texttt{l} is \texttt{bv a} > \texttt{bv b}, or equivalently \texttt{a} ∧ ∼\texttt{b}. It
follows that the hardware for line \texttt{l} requires an inverter for \texttt{b} and an and
gate for \texttt{a} and \texttt{bbar} (the output from the inverter). The hardware for \texttt{g} is
9.2. **TECHNIQUES V — THEOREMS ON THE FLY**

![Diagram](image)

**Figure 9.2 Comp implementation**

symmetric. To derive hardware for e we may note that the values on the three output lines are mutually exclusive so that \((a = b) = (\sim 1 \land \sim g) = \sim (1 \lor g)\), that is, the hardware for the line e simply nor’s the other two outputs. The implementation is a direct wiring of these derivations.

```plaintext
#let Comp_imp = new_definition
('Comp_imp',
  "Comp_imp a b l e g
   = ? abar bbar .
     (inv a abar) \/
     (inv b bbar) \/
     (and2_imp abar b l) \/
     (nor2 l g e) \/
     (and2_imp a bbar g)
   ").;;
Comp_imp =
|- ! a b l e g.
  Comp_imp a b l e g =
  (? bbar bbar.
    inv a abar \/
    inv b bbar \/
    and2_imp abar b l \/
    nor2 l g e \/
    and2_imp a bbar g)
```

9.2.2 **Verification**

The first steps in the proof are straightforward: we set the goal, strip away the quantified variables, rewrite for the primitives, and then eliminate existentially quantified variables. The proof development is done in stages simply to reinforce how we use correctness statements to replace implementations by specifications.
To eliminate the two occurrences of \texttt{and\_imp}, we could rewrite with its definition in terms of \texttt{nand2} and \texttt{inv}, but in effect that strategy would simply flatten the design. Having invested some work in combining these two parts and finding a nice mathematical representation of their composition, it is foolish to throw it away. We first rewrite with the correctness statement \texttt{and\_correct} which tells us (informally) that \texttt{and\_imp = and\_spec}

and then rewrite with \texttt{and\_spec}
We have now replaced all the “hardware” by provenly equivalent specifications and our goal is “pure logic”. The next (standard) step is to remove the hidden lines.

On the left hand side of the goal are the predictions made by our design for the values on the three output lines, and on the right hand side are the predictions made by our specification. The relations for \( l \) identical on both sides, likewise for \( g \), and we “cancel” them using a very useful tactic devised by Tom Melham called \textsc{cancel} \textsc{conj} \textsc{tac} (listed in appendix B).

\textsc{cancel} \textsc{conj} \textsc{tac} can be used when the goal is single relation with conjuncts in common on both sides. The conjuncts may be in jumbled order, as above. Notice that the common terms are \textit{not} discarded but placed on the assumption list — after all, they may turn out to be false. What is left may be solved by bool-casing on \( a \) or \( b \).
9.3 **bitComp**

In the next section we specify and verify a word ripple comparator\(^2\) constructed from a row of bitComp devices. In this section, we describe the basic algorithm and specify and verify the bitComp device.

Consider the bit patterns below, which if so interpreted, would represent the numbers 22 and 17 respectively.

---

\(^2\)Remember that we are working with nums (non-negative integers).
To decide whether one is greater, equal, or less than another, we consider the bits in turn working from the most significant end (here bit 4). No decision can be made whilst the bits are equal (here bits 4 and 3). But as soon as we reach bits that are not equal (bits 2), then their individual comparison gives the result of the word comparison and the remaining bits have no effect on the result (bits 1 and 0). If, on reaching the end of the word, all the bits have been found to be pairwise equal, then the words are equal.

![Diagram of word comparator](image)

We already have a device \texttt{comp} that can compare two bits. In order to build a word comparator we wire together a row of slightly more powerful \texttt{bitComp} devices, as sketched in figure 9.3, with control lines named \texttt{gsf} (greater so far, initially false), \texttt{esf} (equal so far, initially true), and \texttt{lsf} (less so far, initially false). Control ripples through from the high order end to the low order end. At stage $k$, if $\texttt{esf}_k = \texttt{F}$, then the choice has been made by a more significant bit-pair, and the three control signals are passed through. If $\texttt{esf}_k = \texttt{T}$, then the decision can be made locally by a \texttt{comp} device. A \texttt{bitComp} is thus an extended \texttt{comp} device.
9.3.1 Specification

As depicted in figure 9.4 the circuit expects two input signals a and b for local comparison. The three signals coming in from the left lsf, esf and gsf are concerned with decisions taken amongst more significant bits (if any). The specification relates values on l, e and g to the five input signals. The easiest case to consider is that of e. This output signal will be high only if the more significant bits are all equal (signalled by esf = T) and the local bits are equal. Hence we can write e = esf ∧ (bv a = bv b). The value on l will be high if the decision is already made lsf = T or if the more significant bits were all equal and locally a = F and b = T. This we write as l = lsf ∨ (esf ∧ (bv a < bv b)). The expression for g is similar.

9.3.2 Consequences of the specification

One and only one of lsf, esf and gsf is to be high. The device emits three signals l, e and g one and only one of which of which is to be high. Is this true? Let’s find out.
Two worthwhile and cheap tests on the specification are to see what it predicts when the value on \texttt{esf} is known. When \texttt{esf} = \texttt{F}

\begin{verbatim}
#REWRITE_RULE []
(SPEC [ "a:bool"; "b:bool"; "l:bool"; "F"; "esf:bool" ]
     bitComp_spec );

|- !l e g.
     bitComp_spec a b l F esf l e g = (l = lsf) \land \neg e \land (g = esf)
\end{verbatim}

then either \texttt{lsf} or \texttt{gsf} (but not both) will be true, and that will be the only true value to be propagated. When \texttt{esf} = \texttt{T} (then both \texttt{lsf} and \texttt{gsf} will be low)

\begin{verbatim}
#REWRITE_RULE []
(SPEC [ "a:bool"; "b:bool"; "F"; "T"; "F" ]
     bitComp_spec );

|- !l e g.
     bitComp_spec a b F T F l e g =
         (l = (bv a) < (bv b)) \land
         (e = (bv a = bv b)) \land
         (g = (bv a) > (bv b))
\end{verbatim}

then the device will make the decision locally and only one high value will be output.

We may also prove a theorem similar in statement and proof to \texttt{lem1}.

\begin{verbatim}
!a b lsf esf gsf l e g .
     (bitComp_spec a b lsf esf gsf l e g \land oneAssereted lsf esf gsf)
     ==> oneAssereted l e g;

!a b lsf esf gsf l e g .
     bitComp_spec a b lsf esf gsf l e g \land oneAssereted lsf esf gsf ==> oneAssereted l e g

() : void
\end{verbatim}

\begin{verbatim}
(REPEAT GEN_TAC
    THEN PURE_REWRITE_TAC [ bitComp_spec; oneAssereted; oneHigh; GREATER ]
    THEN STRIP_TAC);
OK.
3 subgoals
"(1 \lor \neg e \lor g) \land
  ((1 => \neg e) \land (1 => \neg g)) \land
  ((e => \neg g) \land (e => \neg (g => \neg 1))) \land
  ((g => \neg 1)) \land
  (g => \neg e)"
\end{verbatim}
Note again that applying \texttt{STRIP_TAC} has split the complicated antecedent into its constituent parts on the assumption list, making the latter easy to read and convenient to manipulate. Why are there three subgoals? The definition of \texttt{one asserted} contains three terms or 'ed together, thus the antecedent in our original goal has the form...
Writing this as \( \alpha \land ((lsf \lor esf \lor gsf) \land \beta) \) for short, we have to show that the goal holds for \( \alpha \land (lsf \land \beta) \), \( \alpha \land (esf \land \beta) \), and \( \alpha \land (gsf \land \beta) \). If we can prove these three subcases, then we can certainly prove the goal. Check the assumptions and make sure you understand why all the terms are there.

The second thing to notice is that when we do our rewrites from the assumptions, we will first use the equations for \( l, e \) and \( g \) and and express the goal in terms of \( lsf, esf, gsf \), \( bv \ a \) and \( bv \ b \). In the case of the top goal (the other subgoals are similar) we can rewrite with \( lsf \) but none of the other assumptions are directly usable since they are implications. This is a pity because amongst them we find \( lsf \models \neg esf \) and \( lsf \models \neg gsf \). From these and \( lsf \) we may infer that \( \neg esf \) and \( \neg gsf \), and it would be very nice if we could rewrite with them too!

In situations like this, we use \texttt{RES_TAC} which examines the assumption list and augments it by any new terms that it can infer by “resolution”. Typically if the assumption list contains both \( A \) and \( A \models B \), then \( B \) will be added. If the assumption list contains both \( A \) and \( A = B \), then \( B \) will be added. \texttt{RES_TAC} will pattern match. We backup and try again on all three branches.
Note that we are danger of being overwhelmed by the sheer number of assumptions, most of which we will not use. We will show you how to keep better control over the size of the assumption list in later examples. The pleasant surprises are that two cases have been solved and that the goal is the same as that for *lem1* at this stage and so we have no more real work to do. Here is the complete proof:
9.3. BITCOMP

9.3.3 Implementation

Again a reasonable implementation can be inferred from the specification. We leave the pleasure of deriving it to you.
9.3.4 Verification

The first steps in the verification are obvious: rewrite and then remove the hidden line equations.

```plaintext
#g "! a b lsf esf gsf l e g.
  bitComp_imp a b lsf esf gsf l e g
      = bitComp_spec a b lsf esf gsf l e g";;
"! a b lsf esf gsf l e g.
  bitComp_imp a b lsf esf gsf l e g = bitComp_spec a b lsf esf gsf l e g"
()
```

At this juncture we note that the line equations are respectively jumbled versions of the same terms. This suggests that we reorder terms on the left hand side so that they follow the order on the right. Here the appropriate theorems are

```plaintext
#( CONJ_SYM, DISJ_SYM );
  (l = (bv a) < (bv b) \/ esf \/ lsf) /
  (e = (bv a = bv b) \/ esf) /
  (g = (bv a) > (bv b) \/ esf \/ gsf) =
  (l = lsf \/ esf \/ (bv a) < (bv b)) /
  (e = esf \/ (bv a = bv b)) /
  (g = gsf \/ esf \/ (bv a) > (bv b))"
()
```

Our first attempt fails.
We cannot rewrite (even once) as rewriting uses pattern matching and this rewrite will be applied on both sides of the goal. To get round this we specialise \texttt{DISJ/SYM} twice as below

\begin{verbatim}
  #e(PURE_ONCE_REWRITE_TAC [ DISJ_SYM ]);;
  OK.
  "(l = lsf \ (bv a) \ (bv b) /\ esf) /
   (e = (bv a = bv b) /\ esf) /
   (g = gsf \ (bv a) > (bv b) /\ esf) =
   (l = esf \ (bv a) < (bv b) /\ lsf) /
   (e = esf \ (bv a = bv b)) /
   (g = esf \ (bv a) > (bv b) \ gsf)"

() : void
\end{verbatim}

If we rewrite with these theorems the presence of \texttt{bv} will ensure that we pattern match only on the left hand side of the goal, but having specified exactly what we want, we might as well use substitution which is much faster than rewriting since it attempts no pattern matching.

\begin{verbatim}
  #SPECL [ "(bv a < bv b) /\ esf"; "lsf:bool" ] DISJ_SYM ;
  |- (bv a) < (bv b) /\ esf \ lsf = lsf \ (bv a) < (bv b) /\ esf

  #SPECL [ "(bv a > bv b) /\ esf"; "gsf:bool" ] DISJ_SYM ;
  |- (bv a) > (bv b) /\ esf \ gsf = gsf \ (bv a) > (bv b) /\ esf
\end{verbatim}

\begin{verbatim}
  #b();;
  "(l = (bv a) < (bv b) /\ esf \ lsf) /
   (e = (bv a = bv b) /\ esf) /
   (g = (bv a) > (bv b) /\ esf \ gsf) =
   (l = lsf \ esf \ (bv a) < (bv b)) /
   (e = esf \ (bv a = bv b)) /
   (g = gsf \ esf \ (bv a) > (bv b))"

() : void

#SUBST_TAC ;;
- : (thm list -> tactic)
\end{verbatim}
All that remains is for us to rotate the conjunctions on the left and we have a reflection.

Here is the theorem in tidy format:
9.4 Word comparator

![Diagram of nComp]

**Figure 9.5 nComp**

### 9.4.1 Specification

Here, we do not insist that the initial value of (lsf, esf, gsf) be (F, T, F), because we might wish to build a comparator in, say, 4-bit slices. The device takes two words \(a\) and \(b\) as inputs, and outputs three bool values \(e\), \(g\), and \(l\). Informally, \(e\) is high iff \(a = b\), \(g\) is high iff \(a > b\), and \(l\) is high iff...
\( a < b \). The specification of the word comparator is trivial to express now that we have defined the \texttt{val} abstraction. Its structure is identical to that of \texttt{bitComp.spec}. We have merely substituted \texttt{val f n} for \texttt{bv f}: remember that \texttt{bv (f 0) = val f 0}. This reminds us of the value of keeping to a specific style of definition.

```plaintext
#let nComp_spec = new_definition
 (\nComp_spec' ,
 "nComp_spec n a b lsf esf gsf l e g
 = (l = lsf \ (esf \ (val a n < val b n))) /
 (e = esf \ (val a n = val b n)) /
 (g = gsf \ (esf \ (val a n > val b n)))
 ");
 nComp_spec =
|!n a b lsf esf gsf l e g .
nComp_spec n a b lsf esf gsf l e g =
 (l = lsf \ esf \ (val a n < val b n)) /
 (e = esf \ (val a n = val b n)) /
 (g = gsf \ esf \ (val a n > val b n))
```

### 9.4.2 Consequences of the specification

As a quick check we carry out two cheap tests on the specification considered as an \( n \)-bit slice. If \texttt{esf} is false, then the result was found in a previous slice, and the other two decision values should percolate through unchanged (think about it).

```plaintext
#REWRITE_RULE []
 (SPECL [ "n:num" ; "a:num->bool" ; "b:num->bool";
   "lsf:bool" ; "F" ; "gsf:bool"
 ] nComp_spec);
|! l e g .
nComp_spec n a b lsf F gsf l e g = (1 = lsf) \ e \ (g = gsf)
```

If \texttt{esf} is true, then the decision is made within this slice.

```plaintext
#REWRITE_RULE []
 (SPECL [ "n:num" ; "a:num->bool" ; "b:num->bool";
   "F" ; "T" ; "F"
 ] nComp_spec);
|! l e g .
nComp_spec n a b F F l e g =
 (1 = (val a n < (val b n))) /
 (e = (val a n = val b n)) /
 (g = (val a n > (val b n))
```
Another obvious consequence is that if one control input is asserted, then only one output may be asserted. The proof of this property turns out to be exactly that of \textbf{lem2}.

\begin{verbatim}
#let lem3 = prove_thm
('[lem3]',
  "! a b lsf esf gsf l e g.
  (nComp_spec n a b lsf esf gsf l e g
   /\ oneasserted lsf esf gsf)
   \Rightarrow oneasserted l e g",
  REPEAT GEN_TAC
  THEN PURE_REWRITE_TAC
  [ nComp_spec; oneasserted; oneHigh; GREATER ]
  THEN STRIP_TAC THEN RES_TAC
  THEN ASM_REWRITE_TAC
  [ LESS_NOT_EQ; GSYM LESS_NOT_EQ;
    NOT_LESS_EQ; GSYM NOT_LESS_EQ;
    lss_not_rev ]
  THEN MATCH_ACCEPT_TAC
  (PURE_ONCE_REWRITE_RULE [ GREATER ] LorEorG)
);
lem3 =~ ! a b lsf esf gsf l e g.
  nComp_spec n a b lsf esf gsf l e g /\ oneasserted lsf esf gsf \Rightarrow
  oneasserted l e g
\end{verbatim}

9.4.3 Implementation

![Figure 9.6 Word comparator](image)

We use a primitive recursive definition of the implementation, and think of a 1-bit comparator (not shown) as implemented by a single \texttt{bitComp} and an \(n+1\)-bit comparator as being constructed from a \texttt{bitComp} wired onto an \(n\)-bit comparator (shown in figure 9.6).
\#let nComp_imp = new_prim_rec_definition
('nComp_imp',
"(nComp_imp 0 a b lsf esf gsf l e g
 = bitComp_imp (a 0) (b 0) lsf esf gsf l e g)
/
(nComp_imp (SUC n) a b lsf esf gsf l e g
 = ? l1 el g1.
 (bitComp_imp (a (SUC n)) (b (SUC n)) lsf esf gsf l1 el g1)
/
(nComp_imp n a b l1 el g1 l e g))
");;
nComp_imp =
|- (!a b lsf esf gsf l e g.
 nComp_imp 0 a b lsf esf gsf l e g =
 bitComp_imp(a 0)(b 0)lsf esf gsf l e g) /
 (!a b lsf esf gsf l e g.
 nComp_imp(SUC n)a b lsf esf gsf l e g =
 (?l1 el g1.
 bitComp_imp(a(SUC n))(b(SUC n))lsf esf gsf l1 el g1) /
 nComp_imp n a b l1 el g1 l e g))

## 9.4.4 Verification

We start the proof by setting the goal, applying \texttt{INDUCT_TAC} (on \texttt{n}), stripping away the remaining quantifications and rewriting. We need to use \texttt{ASM.REWRITE_TAC} in order rewrite with the induction hypothesis in the inductive step. The tactic is powerful enough to solve the base case directly.
We now remove the hidden lines, standardise the goal by rewriting with \texttt{GREATER} and then effect some trivial simplifications by rewriting with \texttt{bvLss} and \texttt{bvEqI}.
We notice that esf occurs frequently and that the whole goal is trivially satisfied when esf = F. It is thus worthwhile bool-casing on esf.
Exponential sub-terms like \((2 \exp (\text{SUC} \ n)) \times (\text{bv} (\text{a}(\text{SUC} \ n)))\) occur frequently in verifications of arithmetic hardware and can be dealt with quite easily. The standard move is to bool-case, here on \(\text{a}(\text{SUC} \ n)\) and \(\text{b}(\text{SUC} \ n)\). When we do so, the exponential sub-terms will simplify either to \((2 \exp (\text{SUC} \ n)) \times (\text{bv} \ F)\) or to \((2 \exp (\text{SUC} \ n)) \times (\text{bv} \ T)\). Rewriting with \text{bvals} changes these to \((2 \exp (\text{SUC} \ n) \times 0)\) or \((2 \exp (\text{SUC} \ n)) \times (\text{SUC} \ 0)\) respectively. Further rewriting with \text{MULT_CLAUSES} reduces them further to \(0\) or \(2 \exp (\text{SUC} \ n)\) respectively. A final rewrite with \text{ADD_CLAUSES} simplifies any \(+ 0\) or \(0 +\) terms arising.
\textbf{CHAPTER 9. COMPARATORS}

\#bvals;
\texttt{l- (bv \texttt{T} = \texttt{SUC 0}) \langle (bv \texttt{F} = \texttt{0})}

\#MULT_CLAUSES;
\texttt{l- !m n.}
\texttt{(0 \ast m = 0) \langle}
\texttt{(m \ast 0 = 0) \langle}
\texttt{(1 \ast m = m) \langle}
\texttt{(m \ast 1 = m) \langle}
\texttt{((SUC m) \ast n = (m \ast n) + n) \langle}
\texttt{(m \ast (SUC n) = m + (m \ast n))}

\#ADD_CLAUSES;
\texttt{l- (0 + m = m) \langle}
\texttt{(m + 0 = m) \langle}
\texttt{((SUC n) + n = SUC(m + n)) \langle}
\texttt{(m + (SUC n) = SUC(m + n))}

\#e\texttt{(MAP\_EVERY BOOL\_CASES\_TAC)
[ "(a:num\rightarrow\texttt{bool})(SUC n)"; "(b:num\rightarrow\texttt{bool})(SUC n)"
]
THEM \texttt{REWRITE\_TAC [ bvals; ADD\_CLAUSES; MULT\_CLAUSES ]};
OK...
3 subgoals
"1 /\ "c \langle (g = gsf) =
(1 = \texttt{lsf}) \langle (\texttt{val a n} < ((\texttt{val b n}) + (2 \times (\texttt{SUC n})))) \langle
(e = (\texttt{val a n} = (\texttt{val b n}) + (2 \times (\texttt{SUC n})))) /\)
(g = gsf) \langle ((\texttt{val b n} + (2 \times (\texttt{SUC n}))) < (\texttt{val a n})).
[ "!a b lsf esf gsf 1 e = gsf
\texttt{nComp\_imp n a b lsf esf gsf 1 e = gsf
\texttt{nComp\_spec n a b lsf esf gsf 1 e = gsf"

"1 /\ "c \langle g =
(1 = \texttt{lsf}) \langle (\texttt{val a n} + (2 \times (\texttt{SUC n}))) < (\texttt{val b n}) \langle
(e = (\texttt{val a n} + (2 \times (\texttt{SUC n}))) = (\texttt{val b n})) /\)
(g = gsf) \langle ((\texttt{val b n} + (2 \times (\texttt{SUC n}))) = (\texttt{val a n})).
[ "!a b lsf esf gsf 1 e = gsf
\texttt{nComp\_imp n a b lsf esf gsf 1 e = gsf
\texttt{nComp\_spec n a b lsf esf gsf 1 e = gsf"

"1 /\ "c \langle (\texttt{val a n}) < (\texttt{val b n}) \langle
(e = (\texttt{val a n} = \texttt{val b n})) /\)
(g = gsf) \langle (\texttt{val b n}) < (\texttt{val a n}) =
(1 = \texttt{lsf}) \langle ((\texttt{val a n}) + (2 \times (\texttt{SUC n}))) < (\texttt{val b n} + (2 \times (\texttt{SUC n}))) \langle
(e = ((\texttt{val a n}) + (2 \times (\texttt{SUC n}))) = (\texttt{val b n} + (2 \times (\texttt{SUC n})))) /\)
(g = gsf) \langle ((\texttt{val b n} + (2 \times (\texttt{SUC n}))) < (\texttt{val a n}) + (2 \times (\texttt{SUC n}))).
[ "!a b lsf esf gsf 1 e = gsf
\texttt{nComp\_imp n a b lsf esf gsf 1 e = gsf
\texttt{nComp\_spec n a b lsf esf gsf 1 e = gsf"

() : void
9.5 Techniques VI — val and $2^n$

We now take time out to solve a number of commonly-occurring cases involving val a n and 2 EXP (SUC n) that arise in these three sub-goals.

The six cases that arise are dealt with in turn below:

1. $\text{val a n < val b n + 2 EXP (SUC n)} \rightarrow T$.
   The idea here is that since $(\text{val a n}) < (2 \text{ EXP (SUC n)})$, then $(\text{val a n}) < \text{anything + 2 EXP (SUC n)}$. We use modus ponens on two given theorems as follows:

   ```
   #lss_lss_add;;
   | a. x. x < a --> x < (a + b)
   #maxword;;
   | a. (val a n) < (2 EXP (SUC n))
   #let lem0 = MATCH_MP lss_lss_add (SPEC_ALL maxword);;
   lem0 = |- !a. (val a n) < ((2 EXP (SUC n)) + b)
   #let lem1 = PURE_ONCE_REWRITE_RULE [ADD_SYM] lem0;;
   lem1 = |- !b. (val a n) < (b + (2 EXP (SUC n)))
   ```

2. $\text{val a n + 2 EXP (SUC n)} < \text{val b n} \rightarrow F$.
   Amongst the pre-proved theorem in nums we find

   ```
   #lss_not_rev;;
   | a. b. a < b ==> b < a
   ```

   The theorem we are after, which states that $(\text{val a n + anything} < 2 \text{ EXP (SUC n)})$, is an application of modus ponens on lss_not_rev and lem1.

   ```
   #let lem2 = MATCH_MP lss_not_rev (SPEC_ALL lem1);;
   lem2 = |- !b. (val a n) < (b + (2 EXP (SUC n)))
   ```

3. $\text{val a n + 2 EXP(SUC n)} < \text{val b n + 2 EXP(SUC n)} \rightarrow \text{val a n < val b n}.$
   We simply cancel the exponential term by rewriting with the standard theorem LESS_MONO_ADD_EQ.

   ```
   #LESS_MONO_ADD_EQ;;
   |- !m n p. (m + p) < (n + p) = m < n
   ```
4. \( \text{val } a \, n = \text{val } b \, n + 2 \, \text{EXP} \, (\text{SUC} \, n) \rightarrow F. \)

We draw your attention to two useful standard theorems:

[Code example]

\[ \text{lem3 uses modus ponens on the former and lem1.} \]

5. \( \text{val } a \, n + 2 \, \text{EXP} \, (\text{SUC} \, n) = \text{val } b \, n \rightarrow F. \)

Trivial.

[Code example]

6. \( \text{val } a \, n + 2 \, \text{EXP} \, (\text{SUC} \, n) = \text{val } b \, n + 2 \, \text{EXP} \, (\text{SUC} \, n) \rightarrow \text{val } a \, n = \text{val } b \, n. \)

A simple cancellation, this time using \textit{LESS\_MONO\_ADD\_EQ}.

[Code example]

...and back to the top sub-goal:

We remind you of the top sub-goal:
All we need do here is cancel the exponent terms. We are then left with a tautology.

The last two cases are quite trivial—we simply rewrite with selected lemmas.
Here is a tidy version of the proof.
#timer true;;
false : bool
Run time: 0.0s

#let nComp_correct = prove_thm
('nComp_correct',
  " n a b lsf esf gsf g e l1. nComp_imp n a b lsf esf gsf g e l1 = nComp_spec n a b lsf esf gsf g e l1",
INDUCT_TAC THEN REPEAT GEN_TAC
THEN ASM_REWRITE_TAC
[ nComp_imp; nComp_spec;
  bitComp_correct; bitComp_spec; val
 ]
THEN EXISTS_ELIM_TAC
THEN PURE_REWRITE_TAC [ GREATER; bvLss; bvEq ]
THEN BOOL_CASES_TAC "esf:bool" THEN REWRITE_TAC []
THEN MAP_EVERY BOOL_CASES_TAC
[ "a(SUC n):bool"; "b(SUC n):bool" ]
THEN
  let lem0 = MATCH_MP lss_lss_add (SPEC_ALL maxword) in
  let lem1 = PURE_ONCE_REWRITE_RULE [ ADD_SYM ] lem0 in
  let lem2 = MATCH_MP lss_not_rev (SPEC_ALL lem1) in
  let lem3 = MATCH_MP LESS_NOT_EQ (SPEC_ALL lem1) in
  let lem4 = GSYM lem3 in
  REWRITE_TAC
[ bvLss; MULT_CLAUSES; ADD_CLAUSES;
  LESS_MONO_ADD_EQ; EQ_MONO_ADD_EQ;
  lem1; lem2; lem3; lem4
 ]
);

nComp_correct =
|~ n a b lsf esf gsf g e l1. nComp_imp n a b lsf esf gsf g e l1 = nComp_spec n a b lsf esf gsf g e l1

Run time: 19.7s
Garbage collection time: 10.9s
Intermediate theorems generated: 3389
Part IV

ALU case study
Chapter 10

Step 1: ALU overview

In this and the next two chapters we specify and verify the arithmetic and logical unit presented in Lewin [72, pages 81–89]. Since an ALU is a fairly complicated device, we explain its intended behaviour and justify Lewin’s design before attempting a formal specification and a verification. The heart of the implementation, a ripple carry adder, is verified in chapter 11 and its proof constitutes the bulk of the total effort for this case study. The rest of the ALU proof is covered in chapter 12.

Lewin’s ALU takes two word inputs, \( a \) and \( b \), and four control signals \( E, s_0, s_1, \) and \( \text{cin} \). \( a \) and \( b \) are the basic word-long operands and the control signals are used to encode the desired operation over them. \( E \) controls the mode of operation, either arithmetic (\( E = T \)) or logical (\( E = F \)). \( s_0 \) and \( s_1 \) are used to select a particular operation within that mode (see table 10.1 below). \( \text{cin} \) is interpreted as either carry-in or borrow for the arithmetic operations but is irrelevant for the logical operations. The output appears as word \( s \). \( c \) is the carry-out when in arithmetic mode and is always set to \( F \) in logical mode\(^1\). As with the ripple comparator, we can envisage building the ALU in slices with \( \text{cin} \) supplying the carry/borrow between slices.

\(^1\)Lewin leaves \( c \) unspecified in logical mode but it is easy to show that in his implementation it happens to take the value \( a \land b \). We prefer to set \( c = F \) in our design. This is the only change we make to Lewin’s proposal.
Table 10.1 ALU control bit encodings

Here is an informal preliminary specification of Lewin’s ALU. The specification uses an IF-expression to do a major split between modes before enumerating cases.

First design refinement

Lewin’s design is based upon the ripple carry adder. He first notes that an adder can be made to achieve a range of arithmetic operations (increment, decrement, and subtract as well as add) if we modify the `b` input. For example, we can increment `a` by overriding the input on `b` by all false (all bits of `b` set to false) and setting the carry-in `cin = T`.

An adder subsystem can also be made to perform logical (bitwise) operations if we turn off the carry bits between its constituent full adders. This is easy to achieve in practice — we simply “and” the carry into each full adder with the enable signal `E` as shown in figure 10.2. We call the resulting circuit `AX`. 
When \( E = T \), the \( q \) carry into the full adder is \( \text{cin} \) and the \( \text{AX} \) behaves like \( \text{fullAdder} a_j b_j \text{cin} s_j c_j \). When \( E = F \), the \( q \) carry into the full adder is \( F \) and the \( \text{AX} \) behaves like \( \text{fullAdder} a_j b_j F s_j c_j \). From the specification of a 1-bit full adder, we may infer that \( s_j = a_j \oplus b_j \) and \( c_j = a_j \land b_j \) (see exercise 10.3).

We give the name \( \text{nAX} \) to a row of \( \text{AX} \) devices connected together in ripple fashion. When in arithmetic mode \( (E = T) \), the carries are fed through, and the \( \text{nAddXor} \) behaves like an adder sub-system. When in logical mode \( (E = F) \), the carries are not fed through. The required range of logical operations is realised by modifying the \( a \) and the \( b \) inputs appropriately. For example, with \( E = F \) and each bit of \( b \) set to true, \( \text{nAX} \) will invert each bit of \( a \).

Lewin uses \( \text{nAX} \) in his design. In our implementation, we want to force the last carry-out low when in logical mode and this means tacking on an extra \( \text{and2} \) at the end of an \( \text{nAX} \). This we call the \( \text{nAddXor} \).

This first design refinement is shown in figure 10.3 overleaf. It decomposes the ALU into two major sections:

1. the top box, \( \text{nMod} \), uses the \( E \), \( s_0 \), and \( s_1 \) control signals to deduce the mode and case of the operation and emits appropriately modified \( a \) and \( b \) operands.

2. the lower box, \( \text{nAddXor} \), is a modified ripple carry adder which operates as an adder when \( E = T \) and as a row of exclusive or's when \( E = F \).

Second design refinement: arithmetic operations

Implementing the arithmetic operations is straightforward. With \( E = T \), we leave the \( a \) inputs unchanged and modify the \( b \) inputs using \( s_0 \) and \( s_1 \).
as selectors. Lewin suggests passing each bit of b through its own 4-1mux as shown in figure 10.4.

The special lemma \texttt{nAdderLemma} (formally verified in section 11.4.1) tells us that the outputs of an \texttt{nAdder} are related to its inputs by \( \Sigma = \text{val } s n + 2^{(n+1)}(bv c) \) where \( \Sigma = \text{val } a n + \text{val } b n + bv \text{cin} \). In the interests of textual brevity, we take a few liberties with our notation in the rest of this subsection and we use \texttt{a} for \texttt{val } a \texttt{n}, and \texttt{cin} for \texttt{bv cin}, and so on. Recall that \( 0 \leq a, b, s < 2^{n+1} \). Hence \( \Sigma < 2^{(n+1)} \supset c = F \) and \( \Sigma \geq 2^{(n+1)} \supset c = T \).

0. \texttt{nINC n a cin s c.} \hspace{2cm} (s_0, s_1) = (F, F).

The value on a is incremented. We set b to zero (in effect, letting cin do the incrementing). Then \( \Sigma = a + 0 + \text{cin} = a + \text{cin} \).
\* cin = F: \( \Sigma = a \) and no incrementation takes place. The result is \((s, c) = (a, F)\).

\* cin = T: Two sub-cases arise and \( \Sigma = a + 1 \):
  - If \( a < 2^{n+1} - 1 \), then \((s, c) = (a+1, F)\).
  - If \( a = 2^{n+1} - 1 \), then \((s, c) = (0, T)\).

\[
\text{nINC n a cin s c = } (s = \text{carry} \Rightarrow 0 \mid a - \text{cin}) \land \\
(c = \text{carry})
\]
where \( \text{carry} = (a + \text{cin} = \text{MAX}) \)
and \( \text{MAX} = 2^{**(n+1)} \)

1. nDEC n a cin s c. \((s_0, s_1) = (F, T)\).
   The value on \( a \) is decremented. We set \( b \) to all \textit{true}s representing the number \( 2^{n+1} - 1 \). Then \( \Sigma = a + (2^{n+1} - 1) + \text{cin} = 2^{n+1} + (a-1) + \text{cin} \).

\* cin = F: Two sub-cases arise and \( \Sigma = 2^{n+1} + (a-1) \).
  - If \( a = 0 \), then the result is \((s, c) = (2^{n+1} - 1, F)\).
  - If \( a > 0 \), then \((s, c) = (a-1, T)\) is returned.

\* cin = T: \( \Sigma = 2^{n+1} + a \). Hence \((s, c) = (a, T)\).

\[
\text{nDEC n a cin s c = } (s = \text{carry} \Rightarrow \text{MAX} - 1 \mid a - \text{cin}) \land \\
(c = \text{carry})
\]
where \( \text{carry} = (a = 0) \land (c = F) \)
and \( \text{MAX} = 2^{**(n+1)} \)

2. ADD n a b cin. \((s_0, s_1) = (T, F)\).
   a, b and \text{cin} are added.

3. SUB a b cin n. \((s_0, s_1) = (T, T)\).
   The value of \( b \) is subtracted from that of \( a \). \text{cin} is interpreted as an initial borrow. We invert the bits of \( b \). Since \( b + \overline{b} = \text{all \textit{true}s} = 2^{n+1} - 1, \Sigma = a + \overline{b} + \text{cin} = a + ((2^{n+1} - 1) - b) + \text{cin} = 2^{n+1} + (a-b-1) + \text{cin} \).

\* cin = F: Two sub-cases arise and \( \Sigma = 2^{n+1} + (a-(b+1)) \):
  - If \( a \geq b + 1 \), then \((s, c) = (a - (b + 1), T)\).
  - If \( a < b + 1 \), then \((s, c) = ((2^{n+1} + a) - (b + 1), F)\).

\* cin = T: Two sub-cases arise and \( \Sigma = 2^{n+1} + (a-b) \):
  - If \( a \geq b \), then \((s, c) = (a - b, T)\).
  - If \( a < b \), then \((s, c) = ((2^{n+1} + a) - b, F)\).
nSUB n a b cin s c = (s = carry) 
\[ a - (b + \neg cin) \]
\[ | (MAX + a) - (b + \neg cin) | \] 
\[ (c = carry) \]
where carry = (a \geq b + \neg cin) 
and \( \text{MAX} = 2^{**(n+1)} \)

Second design refinement: logical operations

The ALU implements four logical operations: or, negation, logical and, and xor. Lewin takes care to leave the arithmetic modifications to the b input unchanged, and proposes the following extra logic to modify the a inputs.

0. OR n a b. 
\[ (s_0, s_1) = (F, F) \]
Implemented by taking (bitwise) the xor of \( a_j \lor b_j \) and F.

1. NOT n a. 
\[ (s_0, s_1) = (F, T) \]
Implemented by taking (bitwise) the xor of \( a_j \) and T.

2. XOR n a b. 
\[ (s_0, s_1) = (T, F) \]
Trivial.

3. AND n a b. 
\[ (s_0, s_1) = (T, F) \]
Implemented by taking (bitwise) the xor of \( a_j \lor \neg b_j \) and \( \neg b_j \), where \( \neg b_j \) is the inverse of \( b_j \).

The a inputs only need modification when bitwise \( \lor \) or bitwise \( \land \) is selected and we are in logical mode, i.e. in the cases \((E, s_0, s_1) = (F, F, F) \) and \((E, s_0, s_1) = (F, T, F) \) respectively. Per bit of a, all we need is a 4-1 multiplexer, as indicated in figure 10.5, with a little extra logic.
Summary

Here is a tabulation of the design which includes the encoding scheme adopted.

<table>
<thead>
<tr>
<th>Enable</th>
<th>s0</th>
<th>s1</th>
<th>a'_j</th>
<th>b'_j</th>
<th>result</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>a_j</td>
<td>0</td>
<td>a+cin</td>
<td>INC</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>a_j</td>
<td>1</td>
<td>a-cin</td>
<td>DEC</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>a_j</td>
<td>b_j</td>
<td>a+b+cin</td>
<td>ADD</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>a_j</td>
<td>~b_j</td>
<td>a-b-cin</td>
<td>SUB</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>b_j</td>
<td>0</td>
<td>a+b</td>
<td>OR</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>a_j</td>
<td>1</td>
<td>~</td>
<td>NOT</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>a_j</td>
<td>b_j</td>
<td>a⊗b</td>
<td>XOR</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>a_j</td>
<td>~b_j</td>
<td>a∧b</td>
<td>AND</td>
</tr>
</tbody>
</table>

Structure of the proof

Given the theories bits, nums, and words, and the experience gained in previous attempts at an adder verification, the ALU verification we outline in the next two chapters took one person four weeks to complete and polish (there were three complete proof iterations). We proved the general case, valid for ALU’s of all sizes n and for all input combinations. [2, 14] relate the verification (with HOL and with the Boyer-Moore prover respectively) of an ALU of comparable complexity in about the same elapsed time. Notice that exhaustive simulation of the 32-bit version of our ALU would require
16 ⋅ 2^{32} ⋅ 2^{32} tests = 2^{68} \approx 2.9 \times 10^{21}, which works out at about one million cpu years at 100 million trials a second. Whichever theorem prover you use, there would seem to be definite benefits to mechanical verification over simulation at the sub-system level (and above).

The proof was conducted from the bottom up and was partitioned into 5 stages.

1. We started by specifying and verifying the half adder and then the full adder.

2. Next we specified and verified the ripple carry adder. As part of the tests of our specification, we proved that the adder can be made to increment, decrement and subtract.

3. Next we specified and verified the modified adder subsystem (called \texttt{nAddXor}) and showed that it could function as an arithmetic or as a bitwise subsystem.

4. We then specified and verified \texttt{nMod}, the bank of input filters.

5. Finally we completed the hierarchy by wiring together the modified adder and the bank of filters and proved that it met our specification.

Not of all this story is worth relating in its full and gory detail. Instead we focus on the interesting parts of the proof and set the easier parts of the complete proof as exercises. Most new techniques are described in special sections before we enter the proof that requires them.

Chapter 11 is devoted to verifying the adder subsystem (step 2 above). In Chapter 12 we specify and verify \texttt{nAddXor}, the modified adder subsystem, and \texttt{nMod}, the modifier on the a and b busses (steps 3 and 4 above), and elaborate the final stage of the proof (step 5 above). It is interesting to observe that the complexity of the proofs (as measured by the number of inferences required) does not increase as we move up the hierarchy. We suspect that this is due to the good abstraction facilities offered by a higher order logic.
EXERCISES 10

Exercise 10.1 Prove the following properties of bitwise xor.

\[ n_{\text{xor\_sym}} = \neg \neg a \land b \lor \neg a \land \neg b \]
\[ n_{\text{xor\_or\_lemma}} = \neg \neg n \land a \lor \neg n \land \neg a \land (\neg a \land b) \lor (a \land \neg b) \]
\[ n_{\text{xor\_not\_lemma}} = \neg \neg n \land a \lor \neg n \land \neg a \land (\neg a \land \neg b) \land (\neg b \land a) \]
\[ n_{\text{xor\_and\_lemma}} = \neg \neg n \land \neg a \lor \neg n \land \neg a \land \neg (a \lor \neg b) \land \neg (\neg b \lor a) \]

Exercise 10.2 Specify, design and verify the half adder.

Exercise 10.3 Prove

\[ n_{\text{half\_adder\_lemma}} = \neg \neg a \lor \neg \neg c \land \neg b \land (b \lor c) \land (\neg \neg a \lor \neg \neg (b \lor c)) \]

Exercise 10.4 Specify, design and verify the full adder.

Exercise 10.5 Prove

\[ n_{\text{full\_adder\_lemma}} = \neg \neg a \lor \neg \neg c \land \neg b \land (b \lor c) \land (\neg \neg a \lor \neg \neg (b \lor c)) \]

Exercise 10.6 Prove

\[ n_{\text{full\_adder\_as\_xor}} = \neg \neg \text{full\_adder} \land a \land b \land (s = \neg (a \lor b)) \land (s = a \land b) \]

Exercise 10.7 Specify and verify a 4-bit carry lookahead section.
Exercise 10.8 Prove the following 4 lemmata used in our verification of nAdder (chapter 12).

case1 =
| ~ x . (abc < (2 EXP (SUC n)))
  =>> (((x) /\ (val s n = abc))
       = ((val s n + ((2 EXP (SUC n)) * (bv x))) = abc))

case2 =
| ~ x . (abc < (2 EXP (SUC n)))
  =>> (x /\ (val s n = abc)
       = ((val s n + ((2 EXP (SUC n)) * (bv x)))
           = (abc + (2 EXP (SUC n)))))

case3 =
| ~ x . (abc < (2 EXP (SUC n)))
  =>> (((x) /\ (val s n = abc - (2 EXP (SUC n))))
       = ((val s n + ((2 EXP (SUC n)) * (bv x))) = abc))

case4 =
| ~ x . (abc < (2 EXP (SUC n)))))
  =>> (((abc < (2 EXP SUC n))))
  =>> (((x) /\ (val s n = abc - (2 EXP (SUC n))))
       = (val s n + ((2 EXP (SUC n)) * (bv x))) = abc - (2 EXP (SUC n))))
In this chapter, we verify the ripple carry adder subsystem. Correctness proofs of the half adder and the full adder are well within your current compass, and we assume they are available (they were set as exercises 10.2 and 10.4). We introduce three techniques in this chapter: (i) theorem continuations which enable us not to clutter the assumption list with theorems already used and no longer needed, (ii) how to deal with subtraction, and (iii) disjunctive case splitting.

As usual we also prove some consequences of our specifications in order to increase our confidence in them before proving the main result. In this case, we verify a simple consequence of the \texttt{nAdder} specification and then show that by constraining its inputs this adder can be made to subtract, before moving on to our main result.

### 11.1 Techniques V: theorem continuations

So far, when given a goal with conclusion of the form \((x = \text{expr}) \implies C(x)\), we have pushed the antecedent onto the assumption list and then rewritten from the assumptions, as with

```latex
\begin{verbatim}
#x "(x = 0) \implies (x + y = y)";
"(x = 0) \implies (x + y = y)"

() : void

#(DISCH_TAC THEN ASM_REWRITE_TAC []);;
OK.
"0 + y = y"
[ "x = 0"

() : void
\end{verbatim}
```

The antecedent is left on the assumption list and will not be used again. In cases like this, we can do the rewrite and then throw the antecedent away using the \texttt{theorem continuation DISJ THEN}.
Given a goal of the form (asl, a => b), e(DISCH_THEN tac) works as follows.

1. e applies its argument to the goal, so we are really working with DISCH_THEN tac (asl, a => b).

2. DISCH_THEN fails if the conclusion of the goal is not an implication. Otherwise, DISCH_THEN breaks the goal into two parts by assuming the antecedent (ASSUME "a") and reshaping the goal to (asl, b).

3. DISCH_THEN applies tac (ASSUME "a") to the goal (asl, b).

In the example above, we applied SUBST1_TAC (ASSUME "x = 0") to the goal ([], "x + y = y") at step 3. In general, since we have a handle on the antecedent in step 3, we have the opportunity to transform it further if need be. That is, in step 3 above, e(DISCH_THEN (tac o f)) would apply tac(f a) to the goal (asl, b). In the next example, we specialise the antecedent and turn it round before carrying out the substitution.
Here is a “trace” of the call in which we indicate the micro-steps in the call on \texttt{e(DISCH\_THEN \ldots)} as it picks up the antecedent and manipulates it further before returning the new goal.

\begin{verbatim}
e(DISCH\_THEN (SUBST1\_TAC o SYM o SPEC\_ALL))
→ DISCH\_THEN (SUBST1\_TAC o SYM o SPEC\_ALL) ([, "((x.0=x) \implies (x+y=y))"
→ (SUBST1\_TAC o SYM o SPEC\_ALL) (\ldots "!x.0 = x") ([, "x + y = y")
→ (SUBST1\_TAC o SYM) (\ldots "o = x") ([, "x + y = y")
→ SUBST1\_TAC (\ldots "x = 0") ([, "x + y = y")
→ ([, "0 + y = y")
\end{verbatim}

Should the antecedent be in the form of a conjunction of relations, for example "(x = 0) \land (y = 1) \implies (x \ast y = 0) \land (x + y = 1)" and we want to substitute for each relation, we split the antecedent into its individual conjunctions and then effect the substitutions one by one. The appropriate tactic this time is \texttt{DISCH\_THEN \(\text{CONJUNCTS\_THEN SUBST1\_TAC}\)}.

\texttt{CONJUNCTS\_THEN} is a second theorem continuation which allows us to further manipulate the antecedent passed on from \texttt{DISJ\_THEN} before substituting in the goal. In this case, the call on \texttt{DISCH\_THEN} hands over \texttt{ASSUME "(x=0) \land (y=1)"} and a modified goal to its argument. The argument is another theorem continuation \texttt{CONJUNCTS\_THEN}, which is typed as \(\texttt{(thm \rightarrow tactic)} \rightarrow \texttt{(thm \rightarrow tactic)}\), or \texttt{thm\_tactical} for short. \texttt{CONJUNCTS\_THEN tac} takes a continuation of the form \texttt{asm \dashrightarrow p \land q} and forms \texttt{tac (asm \dashrightarrow p) THEN tac (asm \dashrightarrow q)}. Thus this application of \texttt{CONJUNCTS\_THEN} hands over \texttt{SUBST1\_TAC (x=0/\land y=1/-"x=0") THEN SUBST1\_TAC (x=0/\land y=1/-"y=1")} which is then applied to the transformed goal. Here is a trace of this application:

\begin{verbatim}
e(DISCH\_THEN (CONJUNCTS\_THEN SUBST1\_TAC))
→ DISCH\_THEN (CONJUNCTS\_THEN SUBST1\_TAC)
   ([, "(x=0) \land (y=1) \implies (x \ast y=0) \land (x+y=1)"
→ (CONJUNCTS\_THEN SUBST1\_TAC) (\ldots "(x=0)/(y=1)"
→ (SUBST1\_TAC (\ldots "x=0") THEN SUBST1\_TAC (\ldots "y=1")
→ (SUBST1\_TAC (\ldots "y=1") ([, "(0 \ast y=0) \land (0+y=1)"
→ ([, "(0+1=1)"
\end{verbatim}

and here it is in the HOL system
In the case of an antecedent with more than two conjunctions, we may apply `CONJUNCTS_THEN` repeatedly using the iterator `REPEAT_TCL`.

11.2 Techniques VI: dealing with subtraction

There isn’t much built into HOL with regard to subtraction, so a few more handy theorems are proved in the theory `nums` (see appendix C). Here is a selection of the most commonly used theorems for subtraction. The upper case names are built-in; lower case names belong to the theory `nums`.
11.2. TECHNIQUES VI: DEALING WITH SUBTRACTION 223

\[
\begin{align*}
\text{SUB} &= |- \ (\ m \ . \ 0 - n = 0) /\ \notag\\
\text{SUB}_0 &= |- \ ! \ m \ . \ (0 - n = 0) /\ \notag\\
\text{SUB}_\text{EQ}_0 &= |- \ ! \ m \ . \ (m - n = 0) = m \leq n \notag\\
\text{SUC}_\text{SUB}_1 &= |- \ ! \ m \ . \ (\text{SUC} m) - 1 = m \notag\\
\text{SUB}_\text{ADD} &= |- \ ! \ m \ . \ n \leftarrow m \implies (m - n) + n = m \notag\\
\text{EQ}\_\text{MONO}\_\text{ADD}\_\text{EQ} &= |- \ ! \ m \ . \ p \ . \ (m + p = n + p) = (m = n) \notag\\
\text{nsubn} &= |- \ " \ n \ . \ n - n = 0" \notag\\
\text{sub}\_\text{mono}\_\text{eq} &= |- \ " \ ! \ m \ . \ (\text{SUC} m) - (\text{SUC} m) = n - m" \notag\\
\text{sub}\_\text{same}\_\text{eq} &= |- \ " \ ! \ a \ . \ p \ . \ ((a + p) - (b + p)) = (a - b)" \notag\\
\text{add}\_\text{sub} &= |- \ " \ ! \ m \ . \ (n + m) - m = n" \notag\\
\text{sub}\_\text{add}\_\text{assoc} &= |- \ " \ ! \ a \ . \ c \ . \ ((a - b) - c) = (a - (b + c))" \notag\\
\text{add}\_\text{sub}\_\text{assoc} &= |- \ " \ ! \ a \ . \ c \ . \ c \leftarrow b \implies ((a + b) - c = a - (b - c))" \\
\end{align*}
\]

If we are dealing with constants, then the techniques detailed earlier in chapter 9 on reducing \( C_1 < C_2 \) and \( C_1 = C_2 \) have their obvious analogues. Given a term \( C_1 - C_2 \), we rewrite the constants to their standard SUC form and then repeatedly rewrite with \( \text{sub}_\text{mono}_\text{eq} \) to reduce \( \text{SUC}^n 0 - \text{SUC}^n 0 \) to one of: \( \text{SUC}^k 0 - 0, 0 - 0 \), or \( 0 - \text{SUC}^k 0 \) all of which are further reduced by a rewrite with \( \text{SUB}_0 \).

Three cases arise when the terms we are subtracting are not both constants. Note that since goals must always be of type \( \text{bool} \) we cannot have a term like \( a - b \) as goal. Arithmetic expressions must be compared. In what follows, we use \( = \) as the generic comparison operator and take \( a - b = d \) as the iconic goal. There are three (overlapping) cases to consider:

- If \( a \leq b \) then \( a - b = 0 \) and the goal \( a - b = d \) simplifies down to \( 0 = d \). We first prove \( \text{th1} = |- \ a \leq b \) and then rewrite with

\[
\text{PURE}_\text{ONE}_\text{REWRITE}\_\text{RULE} [ \ \text{SYM}\_\text{RULE} \text{SUB}_\text{EQ}_0 ] \ \text{th1}
\]

where \( \text{SYM}\_\text{RULE} \text{SUB}_\text{EQ}_0 = |- \ ! \ m \ . \ m \leftarrow n = (m - n = 0) \).

- If \( a = b \) then \( 0 = d \). The case is trivial.

- If \( a > b \) we first prove \( \text{th2} = |- \ b \leq a \) and then observe that

\[
\text{MATCH}_\text{MP} \text{SUB}_\text{ADD} \ \text{th2} = |- \ (a - b) + b = a () \notag
\]

We rework the goal into the form \( a = d + b \) by adding \( b \) to both sides of the equality with \( \text{SYM}\_\text{RULE} \text{EQ}\_\text{MONO}\_\text{ADD}\_\text{EQ} \) to get \( (a - b) + b = d + b \) and then rewrite with \( () \) which returns \( a = d + b \) as the new goal.

We have found \( \text{SUB}_\text{ADD}, \text{sub}_\text{add}\_\text{assoc} \) and \( \text{add}_\text{sub}\_\text{assoc} \) to be useful for reorganising terms into forms which permit cancellation.
11.3 Techniques VII: disjunctive case splits

In chapter 5 we introduced the craft of specification using the 1-bit full adder and the ripple carry adder. One of the points we made when rationalising our val abstraction was that the outputs depend just upon the sum of the individual input values. In the example below we show you another way of case splitting which can be very useful in reducing the number of cases in arithmetic proofs. Given the specification of a full adder:

```plaintext
#fullAdder_spec;;
|!a b cin s c.
    fullAdder_spec a b cin s c =
    (let sum = (bv a) + ((bv b) + (bv cin))
     in
       ((bv s = (sum < 2 => sum | sum - 2)) \ (c = ~sum < 2)))
```

we seek to prove the following goal (set as exercise 10.5):

```plaintext
#g " ! a b cin s c . fullAdder_spec a b cin s c 
    => ( (bv a + bv b + bv cin) = (bv s + (2*(bv c)))";;
"!a b cin s c.
    fullAdder_spec a b cin s c =>
      ((bv a) + ((bv b) + (bv cin)) = (bv s) + (2 * (bv c)))"
```

The first steps are automatic:

```plaintext
#e(REPEAT GEN_TAC
  THEN PURE_REWRITE_TAC [ fullAdder_spec ]
  THEN let_TAC);;;
OK .
"(bv s =
  ((bv a) + ((bv b) + (bv cin))) < 2 =>
  (bv a) + ((bv b) + (bv cin))
  ((bv a) + ((bv b) + (bv cin)) - 2)) /
  (c = ~(bv a) + ((bv b) + (bv cin)) < 2) =>
  (bv a) + (bv b) + (bv cin)) = (bv s) + (2 * (bv c)))"
```

We now put our new found skill with theorem continuations to use:
Now comes the crunch. If we now bool-case on \(a\), \(b\) and \(cin\), 8 sub-goals arise. When we were arguing for our specification of this device in chapter 5, we noted that individual input values don't matter, only their sum. That intuition is reflected in the current goal. Why not use it? Amongst the theorems proved in the theory \texttt{nums} we find

\begin{verbatim}
#bit3Cases
|- a b cin.
 ((bv a) + ((bv b) + (bv cin)) = 0) \/
 ((bv a) + ((bv b) + (bv cin)) = (SUC 0)) \/
 ((bv a) + ((bv b) + (bv cin)) = (SUC(SUC 0))) \/
 ((bv a) + ((bv b) + (bv cin)) = (SUC(SUC(SUC 0))))
\end{verbatim}

We apply the tactic \texttt{STRIP_ASSUME_TAC} to a specialised version of this theorem. \texttt{STRIP_ASSUME_TAC} takes its theorem argument and simplifies it before adding the simplifications to the assumption list. If its argument is in the form of a set of conjuncts, it breaks them up and places them on the assumption list separately (this often facilitates rewriting). If its argument is a set of disjuncts, it breaks them up into cases and makes us prove the goal for each case. This is just what we need: 4 separate cases, each with an explicit assumption, e.g. "\((bv a) + ((bv b) + (bv cin)) = 0\)", which can be used directly as a rewriting agent.
CHAPTER 11. STEP 2: THE ADDER SUB-SYSTEM

#e(STRIP_ASSUME_TAC
    (SPECIAL [ "a:bool", "b:bool", "cin:bool" ] bit3Cases)
    THEN ASM_REWRITE_TAC []);
    OK.

4 subgoals
"SUC(SUC(SUC 0)) =
    ((SUC(SUC(SUC 0))) < (SUC(SUC 0)) =>
     SUC(SUC(SUC 0))) |
    (SUC(SUC(SUC 0))) - (SUC(SUC 0)) +
    ((SUC(SUC 0)) * (bv~(SUC(SUC 0))) < (SUC(SUC 0))))"[
    "(bw a) + ((bw b) + (bw cin)) = SUC(SUC(SUC 0))"
]

"SUC(SUC 0) =
    ((SUC(SUC 0)) < (SUC(SUC 0)) =>
     SUC(SUC 0)) |
    (SUC(SUC 0)) - (SUC(SUC 0)) +
    ((SUC(SUC 0)) * (bv~(SUC(SUC 0))) < (SUC(SUC 0))))"[
    "(bw a) + ((bw b) + (bw cin)) = SUC(SUC 0)"
]

"SUC 0 =
    (SUC 0) < (SUC(SUC 0)) => SUC 0 | (SUC 0) - (SUC(SUC 0)) +
    ((SUC(SUC 0)) * (bv~0 < (SUC(SUC 0))))"[
    "(bw a) + ((bw b) + (bw cin)) = 0"
]

() : void

It is good to have only 4 cases instead of the 8 cases that would arise from
bool-casing. Not only do the case have a nice physical intuition, but the
style also appreciably reduces the machine time required to carry out the
proof. The rest of the proof is trivial. Compare with your solution to
exercise 10.5.

11.4 Verification of nAdder

The specification developed in the introduction is repeated here as a re-
mindder.
Before proceeding to define a ripple carry implementation, we test our specification in two ways. First we prove a simple lemma which makes nice use of theorem continuations and subtraction. Then we show that the n-bit adder can behave as an n-bit subtracter. The second lemma is a good test of our skill at subtraction. We leave the easier cases (including showing that the adder can be made to behave like an incrementer or a decrementer) as exercises.

### 11.4.1 nAdder specification: test I

We prove the following lemma which was set as an exercise in chapter 5.

```ml
# g "! n a b cin s c .
    nAdder_spec n a b cin s c =>
    ((val a n) + (val b n) + (bv cin)) =
    (val s n) + ((2 EXP (SUC n)) * (bv c))"

(* ! n a b cin s c .
   nAdder_spec n a b cin s c =>
   ((val a n) + (val b n) + (bv cin)) =
   (val s n) + ((2 EXP (SUC n)) * (bv c))")
```

The proof structure is quite simple. When rewritten with `nAdder_spec`, the antecedent is a conjunction of relations for `val s n` and `c`. We substitute for these relations in the consequent, and then case-split. The opening steps are.

```ml
# let nAdder_spec = new_definition

(let nAdder_spec =
  nAdder_spec n a b cin s c
  = let sum = val a n + val b n + bv cin in
    let MAX = 2 EXP (SUC n) in
    ( (val s n = sum < MAX => sum | sum - MAX) \/
      (c = ~(sum < MAX))
  )
);;

nAdder_spec = ...
```
CHAPTER 11. STEP 2: THE ADDER SUB-SYSTEM

At this stage we would like to substitute the left hand side equations for `val s n` and `c` into the right hand side equations. Once the substitution has been carried out, we have no further use for the equations and might as well discard them. We use theorem continuations to pick up the antecedent, split it into two relations and then effect the substitutions.

leaving no clutter on the assumption list. We now do a case split on the IF condition.¹ It turns out that we need to remember which case we are in, so we use `ASM_CASES_TAC` which places a reminder on the assumption list.

¹There is a good physical intuition behind the case split; the condition is true when carry out is high.
One case is proved. As for the other, we use one of the subtraction techniques of section 11.2. The theorems we need are both built-in and all we need are a couple of applications of IMP_RES_TAC and a final rewrite.

```
###OUT_LESS;;
|~ !m n.":"m < n = n <= m

###SUB_ADD;;
|~ !m n. n <= m ==> ((m - n) + n = m)

#e(IMP_RES_TAC NOT_LESS
   THEN IMP_RES_TAC SUB_ADD);;
OK...
"(val a n) + ((val b n) + (bv cin)) =
(((val a n) + ((val b n) + (bv cin))) - (2 EXP (SUC n))) +
(2 EXP (SUC n))"
[ ""(val a n) + ((val b n) + (bv cin))) < (2 EXP (SUC n))"
[ "(2 EXP (SUC n)) <= ((val a n) + ((val b n) + (bv cin)))"
[ "((val a n) + ((val b n) + (bv cin))) - (2 EXP (SUC n))) +
(2 EXP (SUC n)) =
(wal a n) + ((wal b n) + (bv cin)))"

() : void

#e(ASM_REWRITE_TAC []);;
OK..

goal proved

% <***** trace omitted ***** >>%

|~ !n a b cin s c.
   nadder spec a b cin s c ==>
   (((val a n) + ((val b n) + (bv cin)) =
   (val s n) + ((2 EXP (SUC n)) * (bv c)))

Previous subproof:
goal proved
() : void
```

Here is the proof in tidy form:
### 11.4.2 nAdder specification: test II

We now specify an n-bit subtracter and prove that subtraction can be implemented on an adder.

```plaintext
#let nSUB = new_definition
('nSUB', "! n a b cin s c .
  nSUB n a b cin s c = (val s n + ((2 EXP (SUC n)) * (bv c)))
```

In this proof we delay bool-casing and proceed by rewriting where possible making use of further techniques and theorems for subtraction. We start by setting the goal and unfolding the three definitions it contains.
# n a c i n s c. nadder_spec n a (n^not b) cin s c = nSUB n a b cin s c ";
"! n a c i n s c. nadder_spec n a (n^not b) cin s c = nSUB n a b cin s c"

() : void

#val n^not;

|= !n b. val(n^not b)n = (2 EXP (SUC n)) - (SUC(val b n))

#e(REPEAT GEN_TAC
  THEN PURE_REWRITE_TAC [ nAdder_spec; nSUB; valn^not ]
  THEN let_TAC );;

OK..

"(val s n =
  (((val a n) + (((2 EXP (SUC n)) - (SUC(val b n))) + (bv cin))) <
    (2 EXP (SUC n)) =>
    (val a n) + (((2 EXP (SUC n)) - (SUC(val b n))) + (bv cin)) |
    ((val a n) + (((2 EXP (SUC n)) - (SUC(val b n))) + (bv cin))) -
    (2 EXP (SUC n))))) /\n
(c =
"(~(val a n) + (((2 EXP (SUC n)) - (SUC(val b n))) + (bv cin))) <
    (2 EXP (SUC n))) =
(val s n =
  (((val a n) < ((val b n) + (bv^cin))) =>
    (val a n) - ((val b n) + (bv^cin)) |
    ((2 EXP (SUC n)) + (val a n)) - ((val b n) + (bv^cin))) /\n
(c = "(~(val a n) < ((val b n) + (bv^cin)))"

() : void

Simplifying the IF condition

We first rearrange the IF condition on the left into a form where we can cancel the exponential terms. The steps in the transformation are

val a n + (2 EXP (SUC n) - SUC(val b n)) + bv cin < 2 EXP (SUC n)

→ adding SUC(val b n) to both sides

val a n + (2 EXP (SUC n) - SUC(val b n) + SUC(val b n)) + bv cin
  < 2 EXP(SUC n) + SUC(val b n)

→ cancelling SUC(val b n) since SUC(val b n) ≤ 2 EXP(SUC n)

val a n + 2 EXP (SUC n) + bv cin < 2 EXP (SUC n) + SUC(val b n)

→ and finally cancelling the exponential terms

val a n + bv cin < SUC(val b n)
CHAPTER 11. STEP 2: THE ADDER SUB-SYSTEM

Here are these steps in HOL. We start by getting the bracketing right using \texttt{ADD\_ASSOC} and then move the term \texttt{bv\_cin} the extreme left throughout. This leaves the term $2 \cdot \text{EXP} (\text{SUC} n) - \text{SUC} (\text{val} b n)$ at the right end of any subterm containing it, which is just where we want it.

\begin{verbatim}
#e(PURE\_REWRITE\_TAG [ ADD\_ASSOC ] THEN PURE\_ONCE\_REWRITE\_TAG [ ADD\_SYM ]); ;
OK.
"(val s n =
 ((\text{bv\_cin} + ((\text{val a n}) + ((2 \cdot \text{EXP} (\text{SUC} n)) - (\text{SUC} (\text{val} b n)))))) <
 (2 \cdot \text{EXP} (\text{SUC} n)) \Rightarrow
 (\text{bv\_cin} + ((\text{val a n}) + ((2 \cdot \text{EXP} (\text{SUC} n)) - (\text{SUC} (\text{val} b n)))))) |
 ((\text{bv\_cin} + ((\text{val a n}) + ((2 \cdot \text{EXP} (\text{SUC} n)) - (\text{SUC} (\text{val} b n)))))) -
 (2 \cdot \text{EXP} (\text{SUC} n))) /\)
 (c =
 (\text{bv\_cin} + ((\text{val a n}) + ((2 \cdot \text{EXP} (\text{SUC} n)) - (\text{SUC} (\text{val} b n)))))) <
 (2 \cdot \text{EXP} (\text{SUC} n)) =
 (\text{val s n} =
 ((\text{val a n}) < (\text{bv\_cin} + (\text{val} b n))) \Rightarrow
 (\text{val a n}) - (\text{bv\_cin} + (\text{val} b n)) |
 ((\text{val a n}) + (2 \cdot \text{EXP} (\text{SUC} n)) - ((\text{bv\_cin} + (\text{val} b n)))) /\)
 (c = (\text{val a n}) < (\text{bv\_cin} + (\text{val} b n)))"
()
: void
\end{verbatim}

We now prove two helpful lemmata:

\begin{verbatim}
#let lem0 = SPEC\_ALL (PURE\_ONCE\_REWRITE\_RULE [ LESS\_EQ ] maxword); ;
lem0 = \{ (\text{SUC} (\text{val a n})) < (2 \cdot \text{EXP} (\text{SUC} n)) \}

#let lem1 = prove
("(! a b c. (c \leq b) \Rightarrow (((a + (b - c)) < b) = (a < c))", 
 REPEAT GEN\_TAG THEN STRIP\_TAG
 THEN SUBST\_TAG
 (SPEC\_L "a + (b - c)"; "b: num"; "c: num"
 (GSYM LESS\_MORO\_ADD\_EQ))
 THEN PURE\_ONCE\_REWRITE\_TAG [ GSYM ADD\_ASSOC ]
 THEN IMP\_RES\_TAG SUB\_ADD
 THEN ASM\_REWRITE\_TAG []
 THEN SUBST\_TAG (SPEC\_L "b: num"; "c: num" ] ADD\_SYM
 THEN REWRITE\_TAG [ LESS\_MORO\_ADD\_EQ ]; ;
lem1 = \{ ! a b c. c \leq b \Rightarrow ((a + (b - c)) < b = a < c) \}
\end{verbatim}

\begin{verbatim}
MATCH\_MP lem1 lem0; ;
\{ ! a'.
 ((a' + ((2 \cdot \text{EXP} (\text{SUC} n)) - (\text{SUC} (\text{val a n})))) < (2 \cdot \text{EXP} (\text{SUC} n)) =
 a' < (\text{SUC} (\text{val a n})) \}
\end{verbatim}
After using \texttt{ADD\_ASSOC} to bracket together the terms \texttt{bv cin + val a n} on the left hand side of the goal, we rewrite with a pattern matched consequent of \texttt{lem1}.

\begin{verbatim}
#e(PURE\_REWRITE\_TAC [ ADD\_ASSOC; MATCH\_MP lem1 lem0 ]);;
OK.
"(val s n =
 (((bv cin) + (val a n)) < (SUC(val b n))) =>
  (((bv cin) + (val a n)) + ((2 EXP (SUC n)) - (SUC(val b n)))) -
  ((2 EXP (SUC n)))) /
((c = "((bv cin) + (val a n)) < (SUC(val b n))")
(val s n =
 (((val a n) < ((bv\_cin) + (val b n))) =>
  (val a n) - ((bv\_cin) + (val b n)) |
  ((val a n) + (2 EXP (SUC n)) - ((bv\_cin) + (val b n)))) /
((c = "((val a n) < ((bv\_cin) + (val b n))")
)
: void
\end{verbatim}

Cancelling the relations for c

We now try to relate the conditions on the left and on the right. The required lemma is simple:

\begin{verbatim}
#let lem2 = prove
  (" ! A B cin . (A < (bv\_cin + B)) = ((bv cin + A) < SUC B)",
   REPEAT GEN\_TAC
   THEN BOOL\_CASES\_TAC "cin:bool"
   THEN REWRITE\_TAC [ bvals; ADD\_CLAUSES; LESS\_MONO\_EQ ];;
lem2 = |!\ A B cin . A < ((bv\_cin + B) = ((bv cin) + A) < (SUC B))
\end{verbatim}

After rewriting with \texttt{lem2}, the equations for \texttt{c} cancel.

\begin{verbatim}
#e(PURE\_ONCE\_REWRITE\_TAC [ lem2 ] THEN CANCEL\_CONJ\_TAC);;
OK.
"(val s n =
 (((bv cin) + (val a n)) < (SUC(val b n))) =>
  (((bv cin) + (val a n)) + ((2 EXP (SUC n)) - (SUC(val b n)))) -
  ((2 EXP (SUC n)))) =
((val s n =
 ((("((bv cin) + (val a n)) < (SUC(val b n))")
  (val a n) - ((bv\_cin) + (val b n)) |
  ((val a n) + (2 EXP (SUC n)) - ((bv\_cin) + (val b n))))"
 ["c = "((bv cin) + (val a n)) < (SUC(val b n))"]
)
: void
\end{verbatim}
Since the IF conditions are inverses, we write another simple lemma and turn the right hand IF “inside out”.

```haskell
#let lem3 = prove
("! a b (c:num). ((\a => c | b) => (a => b | c))",
 REPEAT GEN_TAC
 THEN BOOL_CASES_TAC "a:bool"
 THEN ASM_REWRITE_TAC []);
lem3 = |- 'a b c. (((\a) => c | b) => (a => b | c))

#e(PURE_ONCE_REWRITE_TAC [ lem3 ]);;
ok .
"(val s n =
((bv cin) + (val a n)) < (SUC(val b n)) =>
((bv cin) + (val a n)) + ((2 EXP (SUC n)) - (SUC(val b n))) |
((bv cin) + (val a n)) + ((2 EXP (SUC n)) - (SUC(val b n))) -
(2 EXP (SUC n))) =
(val s n =
((bv cin) + (val a n)) < (SUC(val b n)) =>
(val a n) + (2 EXP (SUC n)) - ((bv`cin) + (val b n)) |
(val a n) - ((bv`cin) + (val b n)))"
[ "c = `(bv cin) + (val a n)) < (SUC(val b n))) ]
()
: void
```

**Simplifying the ELSE branch**

We now work on the ELSE branch on the left, cancelling the exponent terms by the following transformations

```
bv cin + val a n + (2 EXP(SUC n) - SUC(val b n)) - 2 EXP(SUC n)

→ rebracket via add_sub_assoc then sub_add_assoc

(bv cin + val a n + 2 EXP(SUC n)) - (SUC(val b n) + 2 EXP(SUC n))

→ cancel the exponent terms using sub_same_eq

(bv cin + val a n) - SUC(val b n)
```

Here is the HOL code:
We are now in fine shape to complete the proof. We bool case on cin and do some obvious rewrites with bvals and ADD_CLAUSES, and with sub_mono_eq to cancel matching SUC terms from constant subtraction.

Here is the proof in tidy form:
CHAPTER 11. STEP 2: THE ADDER SUB-SYSTEM

In the presentation of this theorem, we have grouped a number of applications of `PURE_ONCE_REWRITE_TAC` together. Notice that `MAP_EVERY` requires an argument of type `thm list list`. The `\lambda` factor obviates the need to listify the individual (theorem) arguments.

11.4.3 Verification of the adder sub-system

The specification and implementation definitions of the ripple carry adder are too well known to need repetition. The structure of the proof is quite simple: we induct on the sub-system size \( n \). The base case is trivial. In the general case, we manipulate the goal by gathering together exponent terms and cancelling them where possible before case-splitting.

We start off in obvious fashion. We set the goal, induct on \( n \), and rewrite with the appropriate implementation and specification definitions.

```
#let nAdderSubLemma = prove_thm
('nAdderSubLemma',
  " \! n a cin s c.
    nAdder_spec n a (\lt b) cin s c = nSUB n a b cin s c",
  REPEAT GEN_TAC
  THEN PURE_REWRITE_TAC [ nAdder_spec; nSUB; valn\lt ]
  THEN let_TAC
  THEN MAP_EVERY (\th . PURE_ONCE_REWRITE_TAC [ th ])
    [ ADD_ASSOC; ADD_SYM; ADD_ASSOC;
      MATCH_MP lem1 (SPEC_ALL succValLeqExp); lem2
    ]
  THEN CANCEL_CONJ_TAC
  THEN MAP_EVERY (\th . PURE_ONCE_REWRITE_TAC [ th ])
    [ lem3;
      GSYM(MATCH_MP LESS_EQ_ADD_SUB (SPEC_ALL succValLeqExp));
      GSYM(SUB_PLUS); sub_same_eq
    ]
  THEN BOOL_CASES_TAC "cin:bool"
  THEN REWRITE_TAC [ bvals; ADD_CLAUSES; SUB_MONO_EQ ]
);;

nAdderSubLemma =
|- \! n a cin s c. nAdder_spec n a (\lt b) cin s c = nSUB n a b cin s c
```

(\( n \)) : void
11.4. VERIFICATION OF NADDER

237

#e (INDUCT_TAC THEN REPEAT GEN_TAC
THEN PURE_ASM_REWRITE_TAC
[ nAdder_imp; nAdder_spec; fullAdder_correct; fullAdder_spec; val ]
THEN let_TAC);
OK.
2 subgoals
"(?q.
  (val s n =
   (((val a n) + ((val b n) + (bv cin))) < (2 EXP (SUC n)) =>
   (val a n) + ((val b n) + (bv cin)) |
   (val a n) + ((val b n) + (bv cin)) - (2 EXP (SUC n)))) /
   (q = ~((val a n) + ((val b n) + (bv cin))) < (2 EXP (SUC n))))) /
   (bv (SUC n)) =
   (((bv(a(SUC n))) + ((bv(b(SUC n))) + (bv q))) < 2 =>
   (bv(a(SUC n))) + ((bv(b(SUC n))) + (bv q)) |
   (bv(a(SUC n))) + ((bv(b(SUC n))) + (bv q)) - 2)) /
   (c = ~((bv(a(SUC n))) + ((bv(b(SUC n))) + (bv q)) < 2)) =
   (((val s n) + ((2 EXP (SUC n)) * (bv(s(SUC n))))) +
   (((val a n) + ((2 EXP (SUC n)) * (bv(a(SUC n))))) +
   ((val a n) + ((2 EXP (SUC n)) * (bv(a(SUC n))))) + (bv cin)) <
   (2 EXP (SUC (SUC n))) =>
   ((val b n) + ((2 EXP (SUC n)) * (bv(b(SUC n)))) +
   ((val b n) + ((2 EXP (SUC n)) * (bv(b(SUC n)))) + (bv cin)) |
   ((val a n) + ((2 EXP (SUC n)) * (bv(a(SUC n))))) +
   ((val a n) + ((2 EXP (SUC n)) * (bv(a(SUC n))))) +
   ((val b n) + ((2 EXP (SUC n)) * (bv(b(SUC n)))) + (bv cin)) -
   (2 EXP (SUC (SUC n)))) /
   (c = ~((val a n) + ((2 EXP (SUC n)) * (bv(a(SUC n))))) +
   (((val b n) + ((2 EXP (SUC n)) * (bv(b(SUC n))))) + (bv cin)) <
   (2 EXP (SUC (SUC n))))")
[ "!a b cin s c.
  nAdder_imp n a b cin s c = nAdder_spec n a b cin s c" ]

"(bv(s 0) =
   (((bv(a 0)) + ((bv(b 0)) + (bv cin))) < 2 =>
   (bv(a 0)) + ((bv(b 0)) + (bv cin)) |
   (bv(a 0)) + ((bv(b 0)) + (bv cin)) - 2)) /
   (c = ~((bv(a 0)) + ((bv(b 0)) + (bv cin)) < 2) =
   (bv(s 0) =
   (((bv(a 0)) + ((bv(b 0)) + (bv cin))) < (2 EXP (SUC 0)) =>
   (bv(a 0)) + ((bv(b 0)) + (bv cin)) |
   (bv(a 0)) + ((bv(b 0)) + (bv cin)) - (2 EXP (SUC 0))) /
   (c = ~((bv(a 0)) + ((bv(b 0)) + (bv cin)) < (2 EXP (SUC 0))))" ( : void

Base case. Trivial.
Physical intuition tells us that individual values don’t matter, only their sum. Accordingly we rearrange the exponential terms on the right so that
they are grouped as
\[(\text{val a n} + \text{val b n} + \text{bv cin}) + (2 \times \text{EXP(SUC n)} \times (\text{bv(a(SUC n)}) + \text{bv(b(SUC n)))})\]
and then do cases using `bit2Cases` from the theory `nums`.

```
#bit2Cases;;
| a b.
  (bv a + bv b = 0) /
  (bv a + bv b = SUC 0) /
  (bv a + bv b = SUC(SUC 0))
```

We also need the standard theorem

```
#LEFT_ADD_DISTRIB;;
| m n p. p * (m + n) = (p * m) + (p * n)
```

and the special lemma

```
#let lem0 = prove
  ( "!a b c d e.
    (a + b) + ((c + d) + e) = (a + c + e) + (b + d)",
    INDUCT_TAC THEN REPEAT GEN_TAC
    THEN ASM_REWRITE_TAC [ ADD_CLAUSES; INV_SUC_EQ ]
    THEN SUBST_TAC (SPECL ["c+e"; "b+d"] ADD_SYM)
    THEN MAP_EVERY (\ th.
      PURE_ONCE_REWRITE_TAC [ th ])
    [ SYM_RULE ADD_ASSOC;
      PURE_ONCE_REWRITE_RULE [ ADD_SYM ] EQ_MONO_ADD_EQ;
      ADD_ASSOC;
      EQ_MONO_ADD_EQ
    ]
    THEN MATCH_ACCEPT_TAC ADD_SYM
  );;
lem0 = |- !a b c d e. (a + b) + ((c + d) + e) = (a + (c + e)) + (b + d)
```

We now rewrite with these agents:
and are now in good shape to case split. We choose to split on \((val\ a\ n + val\ b\ n + bv\ cin) < (2\ EXP\ (SUC\ n))\) via \texttt{ASM\_CASES\_TAC} (it helps to have this case on the assumption list) and on the bit values \(a(SUC\ n)\) and \(b(SUC\ n)\) using \texttt{bit2Cases} (to minimise the cases). As an aside, when summed, bit values of \(a(SUC\ n)\) and \(b(SUC\ n)\) tell us if the final result will overflow.

We now argue our way to some major simplifications that can be made in the condition on the right and in the else branch on the right. In practice we first probed the goal with

```lisp
#e(map_every (\th . pure Howe Rewrite_Tac (\th))
  [ lesso; sym_rule left_add_distrib ]);;
OK.
"(7q,
 ((val\ s\ n =
  (((val\ a\ n) + ((val\ b\ n) + (bv\ cin))) < (2\ EXP\ (SUC\ n)) =>
  (val\ a\ n) + ((val\ b\ n) + (bv\ cin))) |
  (((val\ a\ n) + ((val\ b\ n) + (bv\ cin))) - (2\ EXP\ (SUC\ n)))) /
  (q = ~((val\ a\ n) + ((val\ b\ n) + (bv\ cin))) < (2\ EXP\ (SUC\ n)))) /
  (bv(a(SUC\ n)) =
  (((bv(a(SUC\ n))) + ((bv(b(SUC\ n))) + (bv\ q))) < 2 =>
  (bv(a(SUC\ n))) + ((bv(b(SUC\ n))) + (bv\ q))) |
  ((bv(a(SUC\ n))) + ((bv(b(SUC\ n))) + (bv\ q))) - 2)) /
  (c = ~((bv(a(SUC\ n))) + ((bv(b(SUC\ n))) + (bv\ q))) < 2)) =
  ((val\ s\ n) + ((2\ EXP\ (SUC\ n)) * (bv(s(SUC\ n)))) =
  ((((val\ a\ n) + ((val\ b\ n) + (bv\ cin))) +
  ((2\ EXP\ (SUC\ n)) * ((bv(a(SUC\ n))) + (bv(b(SUC\ n)))))) <
  (2\ EXP\ (SUC\ (SUC\ n)))) =>
  ((val\ a\ n) + ((val\ b\ n) + (bv\ cin))) +
  ((2\ EXP\ (SUC\ n)) * ((bv(a(SUC\ n))) + (bv\ (b(SUC\ n)))))) |
  (((val\ a\ n) + ((val\ b\ n) + (bv\ cin))) +
  ((2\ EXP\ (SUC\ n)) * ((bv(a(SUC\ n))) + (bv\ (b(SUC\ n)))))) -
  (2\ EXP\ (SUC\ (SUC\ n)))) /
  (c =
  ~((val\ a\ n) + ((val\ b\ n) + (bv\ cin))) +
  ((2\ EXP\ (SUC\ n)) * ((bv(a(SUC\ n))) + (bv\ (b(SUC\ n)))))) <
  (2\ EXP\ (SUC\ (SUC\ n))))";
"!a\ b\ cin\ s\ c.\n nadder_imp n a b cin s c = nadder_spec n a b cin s c" ]
()
: void
```
11.4. VERIFICATION OF NADDER

but the output is too lengthy and tedious to bear presentation here.

Simplifying the IF condition on the right

The condition in question is

\[(\text{val } a \ n + \text{val } b \ n + \text{bv } \text{cin}) + (\text{2 EXP } (\text{SU}C \ n)) \times (\text{bv } (\text{a } (\text{SU}C \ n)) + (\text{bv } (\text{b } (\text{SU}C \ n)))) < (\text{2 EXP } (\text{SU}C (\text{SU}C \ n)))\]

When we apply the case splitting tactic above, the IF condition takes one of the three forms below:

- \text{bv } (\text{a } (\text{SU}C \ n)) + \text{bv } (\text{b } (\text{SU}C \ n)) = 0. The condition will be rewritten to \text{val } a \ n + \text{val } b \ n + \text{bv } \text{cin} < 2 \text{ EXP}(\text{SU}C \ n)) which is always true as can be seen from the theorem abcLssDbleExp proved in the theory nums.

\[
\text{#abcLssDbleExp};
\text{!n a b cin}.
\]

\[(\text{val } a \ n + (\text{val } b \ n + (\text{bv } \text{cin})) < (2 \text{ EXP } (\text{SU}C (\text{SU}C \ n)))\]

- \text{bv } (\text{a } (\text{SU}C \ n)) + \text{bv } (\text{b } (\text{SU}C \ n)) = 1. The condition will be rewritten to

\[(\text{val } a \ n + \text{val } b \ n + \text{bv } \text{cin}) + 2 \text{ EXP } (\text{SU}C \ n) < 2 \text{ EXP } (\text{SU}C (\text{SU}C \ n))\]

We rewrite the right hand side to \text{2 EXP}(\text{SU}C \ n) + 2 \text{ EXP}(\text{SU}C \ n)

\[
\text{#SPEC } "\text{SU}C \ n" \text{ exp_doubles};
\text{!2 EXP } (\text{SU}C (\text{SU}C \ n)) = (2 \text{ EXP } (\text{SU}C \ n)) + (2 \text{ EXP } (\text{SU}C \ n))\]
and cancel the exponentials on the right using \texttt{LESS\_MONGO\_ADD\_EQ}

- $\text{bv}(a(\text{SUC } n)) + \text{bv}(b(\text{SUC } n)) = 2$. The condition will be rewritten to

\[
(val\ a\ n + val\ b\ n + bv\ \text{cin} + 2\ \text{EXP}\ (\text{SUC } n)) < 2\ \text{EXP}(\text{SUC } n)
\]

which can be shown to be false by rewriting with

\begin{verbatim}
not_add_less;
| "a b. "(a + b) < b
\end{verbatim}

from the theory \texttt{nums}.

**Simplifying the ELSE branch on the right**

\[
(((val\ a\ n) + ((val\ b\ n) + (bv\ \text{cin}))) +
(2\ \text{EXP}\ (\text{SUC } n) * ((\text{bv}(a(\text{SUC } n))) + (\text{bv}(b(\text{SUC } n)))))) -
(2\ \text{EXP}\ (\text{SUC}(\text{SUC } n)))
\]

When we apply the case splitting tactic, the above ELSE branch takes one of the three forms below:

- $\text{bv}(a(\text{SUC } n)) + \text{bv}(b(\text{SUC } n)) = 0$. This case need not be considered since it is never entered. But if it were, it is easy to show from \texttt{abcLzeDoubleExp} that its value must be 0.

- $\text{bv}(a(\text{SUC } n)) + \text{bv}(b(\text{SUC } n)) = 1$. The ELSE branch will be rewritten to

\[
(val\ a\ n + val\ b\ n + bv\ \text{cin} + 2\ \text{EXP}\ (\text{SUC } n) - 2\ \text{EXP}(\text{SUC } n))
\]

We rewrite the rightmost subterm to $2\ \text{EXP}(\text{SUC } n) + 2\ \text{EXP}(\text{SUC } n)$ and cancel one of these exponentials using \texttt{ADD\_SYM sub\_same\_eq}

\begin{verbatim}
PURE_ONCE_REWRITE_RULE [ ADD\_SYM ] sub\_same\_eq;;
| "a b c. (a + b) - (a + c) = b - c
\end{verbatim}

- $\text{bv}(a(\text{SUC } n)) + \text{bv}(b(\text{SUC } n)) = 2$. The ELSE branch will be rewritten to
Further simplifications

Further simplifications can be made on the left hand side amongst the conditions that arise from rewriting the equation for \( bv(s(SUC \_n)) \). We group all the cases that can arise and prove them using the lemma named `demanton` below. This backwards proof does not deserve to be blessed with a more gracious name.

```
#let demanton = prove
   "((SUC \_0) < 2) \land ((SUC \_0) < 2) \land 
   (SUC(s(SUC \_0))) < 2 \land (SUC(s(SUC \_0))) < 2 \land 
   (SUC(s(SUC \_0)) - 2 = 0) \land ((SUC(s(SUC \_0)) - 2 = SUC \_0)"
",
REWRITE_TAC [
   num_CONV "2"; num_CONV "1";
   SUB_MONO_EQ; SUC_NOT; NOT_SUC;
   LESS_MONO_EQ; LESS_0; NOT_LESS_0; SUB_0 ];

```

... back to the main proof

It helps our cause to place a specialised version of `abcLssDbleExp` on the assumption list. It is then automatically called into play when we rewrite from the assumptions and helps dispose of one case.
CHAPTER 11. STEP 2: THE ADDER SUB-SYSTEM

```plaintext
#e(EXISTS_ELIM_TAC
THEN ASSUME_TAC (SPEC_ALL a b c s sSBOOLExp)
THEN ASM_CASES_TAC
  "((val a n) + ((val b n) + (bv cin))) < (2 EXP (SUC n))"
THEN STRIP_ASSUME_TAC
  (SPEC ["a(SUC n):bool"; "b(SUC n):bool"] bit2masses)
THEN ASM_REWRITE_TAC
  [ bvals; MULT_CLAUSES; ADD_CLAUSES; demanton; ivals ]
THEN MAP_EVERY (\th . PURE_ONCE_REWRITE_TAC [ th ])
  [ SPEC "SUC n" exp_doubles;
    PURE_ONCE_REWRITE_RULE [ ADD_SYM ] sub_same_eq;
    LESS_MONO_ADD_EQ; not_add_less; ADD_SUB ]
THEN ASM_REWRITE_TAC [] THEN CANCEL_CONJ_TAC ;;
OK ..
6 subgoals
"((val s n = (val a n) + ((val b n) + (bv cin))) /\ "s(SUC n) =
  ((val s n) + ((2 EXP (SUC n)) * (bv(s(SUC n)))))
  = (val a n) + ((val b n) + (bv cin)))"
[ "\a b cin s c .
  nadder_imp n a b cin s c = nadder_spec n a b cin s c" ]
[ "((val a n) + ((val b n) + (bv cin))) < (2 EXP (SUC n))"
  "(bv(a(SUC n))) + (bv(b(SUC n))) = 0" ]
[ "((val a n) + ((val b n) + (bv cin))) < (2 EXP (SUC(SUC n)))"
  "((val a n) + ((val b n) + (bv cin))) < (2 EXP (SUC n))"
  "(bv(a(SUC n))) + (bv(b(SUC n))) = SUC 0" ]
[ "c" ]

"((val s n = (val a n) + ((val b n) + (bv cin))) /\ "s(SUC n) =
  ((val s n) + ((2 EXP (SUC n)) * (bv(s(SUC n)))))
  = (val a n) + ((val b n) + (bv cin)))"
[ "\a b cin s c .
  nadder_imp n a b cin s c = nadder_spec n a b cin s c" ]
[ "((val a n) + ((val b n) + (bv cin))) < (2 EXP (SUC n))"
  "(bv(a(SUC n))) + (bv(b(SUC n))) = 0" ]
[ "((val a n) + ((val b n) + (bv cin))) < (2 EXP (SUC(SUC n)))"
  "((val a n) + ((val b n) + (bv cin))) < (2 EXP (SUC n))"
  "(bv(a(SUC n))) + (bv(b(SUC n))) = SUC 0" ]
[ "c" ]

"((val s n = (val a n) + ((val b n) + (bv cin))) /\ "s(SUC n) =
  ((val s n) + ((2 EXP (SUC n)) * (bv(s(SUC n)))))
  = (val a n) + ((val b n) + (bv cin)))"
[ "\a b cin s c .
  nadder_imp n a b cin s c = nadder_spec n a b cin s c" ]
[ "((val a n) + ((val b n) + (bv cin))) < (2 EXP (SUC n))"
  "(bv(a(SUC n))) + (bv(b(SUC n))) = 0" ]
[ "((val a n) + ((val b n) + (bv cin))) < (2 EXP (SUC(SUC n)))"
  "((val a n) + ((val b n) + (bv cin))) < (2 EXP (SUC n))"
  "(bv(a(SUC n))) + (bv(b(SUC n))) = SUC 0" ]
```

Although 6 legs show up, there are only 4 distinct cases. Since we have done all the hard work, we leave it to you to complete this verification.

The way we did it was to prove the following 4 lemmata:
which were set as exercises 10.8. The tidy version of the proof reads:

```plaintext
#case 1;;
|- !x.
    abc < (2 EXP (SUC n)) ==> 
    ((val s n = abc) /\ x = 
    (val s n) + ((2 EXP (SUC n)) * (bv x)) = abc))

#case 2;;
|- !x.
    abc < (2 EXP (SUC n)) ==> 
    ((val s n = abc) /\ x = 
    (val s n) + ((2 EXP (SUC n)) * (bv x)) = abc + (2 EXP (SUC n))))

#case 3;;
|- !x.
    ~abc < (2 EXP (SUC n)) ==> 
    ((val s n = abc - (2 EXP (SUC n))) /\ x = 
    (val s n) + ((2 EXP (SUC n)) * (bv x)) = abc))

#case 4;;
|- !x.
    ~((val a n) + ((val b n) + (bv cin))) < (2 EXP (SUC n)) ==> 
    ((val s n = (val a n) + (val b n) + (bv cin)) - (2 EXP (SUC n))) /\ 
    x = 
    (val s n) + ((2 EXP (SUC n)) * (bv x)) = 
    ((val a n) + ((val b n) + (bv cin)) - (2 EXP (SUC n))))
```
11.4. VERIFICATION OF NADDER 247

```
#let nAdder_correct = prove_thm
  (\'nAdder_correct',
   "! n a b cin s c.
    nAdder_imp n a b cin s c = nAdder_spec n a b cin s c",
   INDUCT_TAC THEN REPEAT GEN_TAC
   THEN PURE_ASM_REWRITE_TAC
    [ nAdder_imp; nAdder_spec; 
      fullAdder_correct; fullAdder_spec; val 
   ]
   THEN let_TAC
   THENL
    [ REWRITE_TAC [ EXP; MULT_CLAUSES ] 
    ;
      PURE_ONCE_REWRITE_TAC [ lem0 ]
      THEN PURE_ONCE_REWRITE_TAC [ GSYM LEPT_ADD_DIST ]
      THEN EXISTS_ELIM_TAC
      THEN ASM_CASES_TAC
      "((val n a n) + ((val b n) + (bv cin))) < (2 EXP (SUC n))"
      THEN STRIP_ASSUME_TAC
      (SPEC [ "a(SUC n):bool"; "b(SUC n):bool" ] bit2Cases)
      THEN ASM_REWRITE_TAC
      [ bvals; MULT_CLAUSES; ADD_CLAUSES; demanton; ivals ]
      THEN MAP_EVERY (\ th . PURE_ONCE_REWRITE_TAC [ th ])
       abc.dssDbleExp;
       SPEC "SUC n" exp_doubles;
       sub_same_eq; LESS_MONO_ADD_EQ;
       not_add_less; ADD_SUB 
    ]
   THEN ASM_REWRITE_TAC []
   THEN CANCEL_CONJ_TAC
   THENL
    (map (IMP_RES_TAC o (SPEC "s(SUC n):bool"))
     [ case1; case2; case3; case4; case3 ])
   THEN ASM_REWRITE_TAC []
  )

nAdder_correct =
|- ! n a b cin s c. nAdder_imp n a b cin s c = nAdder_spec n a b cin s c
Run time: 47.7s
Intermediate theorems generated: 7015
```
EXERCISES 11

Exercise 11.1  Given

\[ n\text{INC} = \]
\[ \text{let } n\text{INC} : \text{Nat} \rightarrow \text{Nat} \text{ in } \]
\[ \text{let } n\text{a} = \text{val a} \text{ and } \text{cin} = \text{cin} \text{ in } \]
\[ \text{let } c = 2 \times \text{SUC } n \text{ in } \]
\[ c = \text{cin} \]
\[ \text{prove that } \]

\[ \text{INC\_thm0} = \]
\[ \text{let } n: \text{Nat} \text{ in } \]
\[ \text{let } a: \text{Nat} \text{ and cin: Nat} \text{ in } \]
\[ \text{let } c = \text{cin} \text{ in } \]
\[ \text{prove that } \]

\[ \text{INC\_thm1} = \]
\[ \text{let } n: \text{Nat} \text{ in } \]
\[ \text{let } a: \text{Nat} \text{ and cin: Nat} \text{ in } \]
\[ \text{let } c = \text{cin} \text{ in } \]
\[ \text{prove that } \]

Exercise 11.2  Given the definition

\[ n\text{DEC} = \]
\[ \text{let } n\text{DEC} : \text{Nat} \rightarrow \text{Nat} \text{ in } \]
\[ \text{let } n\text{a} = \text{val a} \text{ and } \text{cin} = \text{cin} \text{ in } \]
\[ \text{let } c = \text{cin} \text{ in } \]
\[ \text{prove that } \]

\[ \text{DEC\_thm0} = \]
\[ \text{let } n: \text{Nat} \text{ in } \]
\[ \text{let } a: \text{Nat} \text{ and cin: Nat} \text{ in } \]
\[ \text{let } c = \text{cin} \text{ in } \]
\[ \text{prove that } \]

\[ \text{DEC\_thm1} = \]
\[ \text{let } n: \text{Nat} \text{ in } \]
\[ \text{let } a: \text{Nat} \text{ and cin: Nat} \text{ in } \]
\[ \text{let } c = \text{cin} \text{ in } \]
11.4. VERIFICATION OF NADDER

Exercise 11.3 Given

\[ nADD = \]
\[ |nADD\ n\ a\ b\ cin\ s\ c = (val\ s\ n =
\begin{align*}
&((val\ a\ n) + ((val\ b\ n) + (bw\ cin))) < (2\ \text{EXP}\ (SUC\ n)) \Rightarrow \\
&(val\ a\ n) + ((val\ b\ n) + (bw\ cin)) | \\
&((val\ a\ n) + ((val\ b\ n) + (bw\ cin))) - (2\ \text{EXP}\ (SUC\ n))) /\ \\
&(c = ~(val\ a\ n) + ((val\ b\ n) + (bw\ cin))) < (2\ \text{EXP}\ (SUC\ n)))
\end{align*}
\]

prove that

\[ ADD_{\text{thm}0} = \]
\[ |-!\ n\ a\ b\ s\ c . \ nADD_{\text{spec}}\ n\ a\ b\ F\ s\ c = (val\ s\ n = (val\ a\ n + val\ b\ n) < (2\ \text{EXP}\ (SUC\ n)) \\
\Rightarrow (val\ a\ n + val\ b\ n) | \\
(((val\ a\ n + val\ b\ n) - (2\ \text{EXP}\ (SUC\ n))) /\ \\
(c = ~(val\ a\ n + val\ b\ n) < (2\ \text{EXP}\ (SUC\ n)))
\]

\[ ADD_{\text{thm}1} = \]
\[ |-!\ n\ a\ b\ s\ c . \ nADD_{\text{spec}}\ n\ a\ b\ T\ s\ c = (val\ s\ n = (SUC(val\ a\ n + val\ b\ n) < 2\ \text{EXP}\ (SUC\ n)) \\
\Rightarrow SUC(val\ a\ n + val\ b\ n) | \\
((SUC(val\ a\ n + val\ b\ n) - (2\ \text{EXP}\ (SUC\ n))) /\ \\
(c = ~(SUC(val\ a\ n + val\ b\ n) < 2\ \text{EXP}\ (SUC\ n)))
\]

Exercise 11.4 Given

\[ nSUB = \]
\[ |-!\ n\ a\ b\ cin\ s\ c . \ nSUB\ n\ a\ b\ cin\ s\ c = (val\ s\ n = ~(val\ a\ n < (val\ b\ n + (bw(~\ cin)))))) \\
\Rightarrow (val\ a\ n - (val\ b\ n + (bw(~\ cin)))) | \\
(((2\ \text{EXP}\ (SUC\ n)) + val\ a\ n) - (val\ b\ n + (bw(~\ cin)))) /\ \\
(c = ~(val\ a\ n < (val\ b\ n + (bw(~\ cin)))))
\]

prove that
CHAPTER 11. STEP 2: THE ADDER SUB-SYSTEM

\begin{align*}
\text{SUB_thm0} &= \neg \! \! n \text{ a b s c} \; \text{. nSUB n a b F s c} \\
&= (\text{val s n} = \neg (\text{val a n} < \text{(SUC(val b n))})) \\
&\Rightarrow (\text{val a n} - \text{(SUC(val b n))}) \\
&\mid ((2 \text{ EXP } \text{(SUC n)}) + \text{val a n}) - (\text{SUC(val b n)})) \lor \\
&\text{(c} = \neg (\text{val a n} < \text{(SUC(val b n))})))
\end{align*}

\begin{align*}
\text{SUB_thm1} &= \neg \! \! n \text{ a b s c} \; \text{. nSUB n a b T s c} \\
&= (\text{val s n} = (\text{val a n} < \text{val b n})) \\
&\Rightarrow (\text{val a n} - \text{(val b n)}) \\
&\mid ((2 \text{ EXP } \text{(SUC n)}) + \text{val a n}) - (\text{val b n})) \lor \\
&\text{(c} = \neg (\text{val a n} < \text{val b n})))
\end{align*}

\begin{align*}
n\text{AdderSubLemmma} &= \neg \! \! n \text{ a cin s c} \\
&n\text{Adder_spec n a (nNot b) cin s c} = \text{nSUB n a b cin s c}
\end{align*}

Exercise 11.5 Specify the n-bit incrementer, decremen ter, and sub-tractor and then prove that

\begin{align*}
\neg \! \! n \text{ a cin s c} \\
&n\text{Adder_spec n a allFalse cin s c} = n\text{Inc_spec n a cin s c}
\end{align*}

\begin{align*}
\neg \! \! n \text{ a cin s c} \\
&n\text{Adder_spec n a allTrue cin s c} = n\text{Dec_spec n a (cIn) s c}
\end{align*}

\begin{align*}
\neg \! \! n \text{ a cin s c} \\
&n\text{Adder_spec n a (nNot b) cin s c} = n\text{Sub_spec n a (cIn) s c}
\end{align*}

Exercise 11.6 Given the HOL definition

\begin{verbatim}
let nALU_spec = new_definition
(\'nALU_spec\',
 "nALU_spec n E s0 s1 cin a b s c
 = E \Rightarrow
% arithmetic operations
  (s0 \& s1) \Rightarrow nINC n a cin s c
 | (s0 \& \neg s1) \Rightarrow nDEC n a cin s c
 | (\neg s0 \& s1) \Rightarrow nADD n a b cin s c
 | (\neg s0 \& \neg s1) \Rightarrow nSUB n a b cin s c
)

% logic operations
  (s0 \& s1) \Rightarrow (nOR n a b s) \lor (c = F))
 | (s0 \& \neg s1) \Rightarrow (nNAND n a b s) \lor (c = F))
 | (\neg s0 \& s1) \Rightarrow (nNOR n a b s) \lor (c = F))
 | (\neg s0 \& \neg s1) \Rightarrow (nXOR n a b s) \lor (c = F))
")
end
\end{verbatim}

prove
11.4. VERIFICATION OF NADDER

\begin{verbatim}

nALUInc =
|\neg \! a \! b \! c \! i\! n \! s \! c .
   nALU_spec n T F F cin a b s c
   = \text{adder_spec n allFalse cin s c}

nALUDec =
|\neg \! a \! b \! c \! i\! n \! s \! c .
   nALU_spec n T F T cin a b s c
   = \text{adder_spec n allTrue cin s c}

nALUAdd =
|\neg \! a \! b \! c \! i\! n \! s \! c .
   nALU_spec n T T F cin a b s c
   = \text{adder_spec n a b cin s c}

nALUSub =
|\neg \! a \! b \! c \! i\! n \! s \! c .
   nALU_spec n T T T cin a b s c
   = \text{adder_spec n a(nNot b) cin s c}

Exercise 11.7  Similarly, prove

nALUOr =
|\neg \! a \! b \! s \! c .
   nALU_spec n F F F cin a b s c
   = (nXOR n (nOr a b) allFalse s) \land \neg c

nALUNot =
|\neg \! a \! b \! c \! .
   nALU_spec n F F T cin a b s c
   = (nXOR n a allTrue s) \land \neg c

nALUXor =
|\neg \! a \! b \! s \! c .
   nALU_spec n F T F cin a b s c
   = (nXOR n a b s) \land \neg c

nALUAnd =
|\neg \! a \! b \! s \! c .
   nALU_spec n F T T cin a b s c
   = nXOR n(nOr a(nNot b))(nNot b)s \land \neg c
\end{verbatim}


Chapter 12

Step 3: ALU verification

We complete the verification of the ALU in this chapter. We first verify the two major sub-components — \texttt{nAddXor}, the modified adder subsystem and \texttt{nMod}, the bank of filters. The final step in the proof is to verify that when wired together these two components implement (our version of) Lewin’s ALU. The proofs turn out to be fairly straightforward and manageable thanks to our hierarchical proof style. Figure 12.1 serves as a map of the structure of the implementation and the sections in which the various proof obligations are carried out.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{ALU_hierarchy.png}
\caption{ALU proof hierarchy}
\end{figure}

The proof structure follows figure 10.2 in that we verify two row circuits — \texttt{nAddXor} and \texttt{nMod} — and wire them together. We could also have organised the proof by first wiring together single \texttt{AX} and \texttt{Mod} devices and then defining a row of these. No special significance should be read into
our organisation — it was enough that we could build upon our previous adder verification in the course of this new proof.

12.1 The nAddXor box

![Diagram of nAddXor](image)

Figure 12.2 The nAddXor

We specify `nAddXor` by

```haskell
#let nAddXor_spec = new_definition
('nAddXor_spec',
 "nAddXor_spec n E a b cin s c
  = (E => nAdder_spec n a b cin s c
    | (nXOR n a b s) \ (c = F))
");;
```

which states simply that when `E` is high, `nAddXor` behaves like an adder; and when `E` is low, `nAddXor` behaves like a bitwise `xor` device on `s` with the `c` output forced low.

As indicated by figure 12.1, we tackle the verification hierarchically, first specifying and verifying `AX`, then `nAX`, and finally `nAddXor`.

12.1.1 Verification of AX

`AX` (figure 10.2) simply wires together an `and2` and a `fullAdder`. The `and2` enables us to set any carry-in to `F`. We start by reminding you of the pertinent definitions
As one might expect, we case split on $E$, and rewrite with the theorem `fullAdderAsXor2` which was set as exercise 10.6.

The only subtlety in the proof lies in not rewriting too soon. After the case split, there are two sub-goals, with $q = \top$ or $q = \bot$ respectively as governing relations for $q$. If we rewrite at once, these terms are simplified to $\neg q$ and $q$. Unfortunately `EXISTS_ELIM_TAC` expects relations of the form $q = \text{expr}$ and is not sensitive enough to handle the first of these cases. The secret is to rewrite purely with `ADD_CLAUSES` as below.

```ml
#fullAdderAsXor2

|- a b c. fullAdder_spec a b F s c = (s = \neg (a = b)) \land (c = a \lor \neg b)
```

```ml
#AND_CLAUSES

|- \!t.
  (T \land t = t) \land
  (t \land T = t) \land
  (F \land t = F) \land
  (t \land F = F) \land
  (t \land t = t)
```
12.1.2 Verification of nAX

nAX is simply a row of AX devices wired together. If E is high, it behaves as an adder; if E is low, it behaves like a row of XORs. This circuit is used in Lewin's design; in effect, he just accepts whatever emerges on carry-out when E = F. We could leave this undefined, as in

```ml
#let nAX_spec' = new_definition
  ("nAX_spec'");
  "nAX_spec' n E cin a b s c
  = (E -> nAdder_spec n a b cin s c
      | nXOR n a b s)
  ");
  nAX_spec' = ...
```

but then we would not be able to complete an equivalence proof. Instead we use

```ml
#let nAX_spec = new_definition
  ("nAX_spec'");
  "nAX_spec n E cin a b s c
  = (E -> nAdder_spec n a b cin s c
      | (nXOR n a b s) /
         (c = a n /
          b n))");
  nAX_spec = ...
```

and the companion implementation definition
We tackle the proof in simple stages. We first prove that when the enable line E is high, the hardware implements an adder. Then we prove that when E is low, the hardware implements a row of xor gates. By simply bool-casing on E we find that we can split the proof into an nAdder proof (which we have done) and an nXOR proof (which is simple). These two correctness statements render the final proof trivial.

nAX adds

The goal we set ourselves is:

```
#let nAX_imp = new_prim_rec_definition
('nAX_imp',
  "nAX_imp O E a b cin s c
   = AX_imp E (a O) (b O) cin (s O) c
   /
   (nAX_imp (SUC n) E a b cin s c
    = ? q .
      (nAX_imp n E a b cin s q) /
      (AX_imp E (n(SUC n)) (b(SUC n)) q (s(SUC n)) c)) ");

nAX_imp = ...
```

```
The first steps are obvious.

```
#e (INDUCT_TAC THEN REPEAT GEN_TAC
  THEN ASM_REWRITE_TAC
  [ nAX_imp; AX_correct; AX_spec ]);;

OK.
2 subgoals

"(\?q.
  nAdder_spec n a b cin s q /
  fullAdder_spec(a(SUC n))(b(SUC n))q(s(SUC n))c) =
  nAdder_spec(SUC n)a b cin s c"
  [ "a b cin s c. nAX_imp n T a b cin s c = nAdder_spec n a b cin s c" ]

"fullAdder_spec(a O)(b O)cin(s O)c = nAdder_spec 0 a b cin s c"

() : void
```

Base case. We could bash out this case by rewriting
If we proceed in this way, it is amusing and fruitful to rewrite in the opposite direction because the implementations are not flattened.

**Induction step.** If we unfold by rewriting with `nAdder_spec`, this case also suffers from the same problem.
Following this path will lead to a lot of rewriting. We backup and rewrite the specifications back to their implementations.

and a simple application of \texttt{REFL\_TAC} will do the trick.
The proof is tidied by folding both induction legs into one.

```
#let nAXLemmaT = prove
  ("! n a b cin s c .
   nAX_imp n T a b cin s c = nAdder_spec n a b cin s c",
   INDUCT_TAC THEN REPEAT GEN_TAC
   THEN ASM_REWRITE_TAC [ nAX_imp; AX_correct; AX_spec ]
   THEN REWRITE_TAC
   [ GSYM nAdder_correct; nAdder_imp; fullAdder_correct ]);

nAXLemmaT =
|! n a b cin s c . nAX_imp n T a b cin s c = nAdder_spec n a b cin s c
Run time: 1.2s
Intermediate theorems generated: 355
```

nAX bitwise xor's

Both the statement of this lemma and its proof are straightforward and need no explanation.

```
#let nAXLemmaF = prove
  ("! n a b cin s c .
   nAX_imp n F a b cin s c = (nXOR n a b s) \ (c = a n \ b n)",
   INDUCT_TAC THEN REPEAT GEN_TAC
   THEN ASM_REWRITE_TAC
   [ nAX_imp; AX_correct; AX_spec; nEq; bvEq; nXOR; nXor; AX_correct ]
   THEN EXISTS_ELIM_TAC
   THEN CANCEL_CONJ_TAC);

nAXLemmaF =
|! n a b cin s c .
   nAX_imp n F a b cin s c = nXOR n a b s /\ (c = a n \ b n)
Run time: 3.1s
Garbage collection time: 0.9s
Intermediate theorems generated: 122
```
12.1. THE NADDXOR BOX

Verifying nAX

The final theorem is now a trivial rewrite.

```
#let nAX_correct = prove_thm
  ('nAX_correct',
   "! E n a b cin s c .
    nAX_imp n E a b cin s c = nAX_spec n E cin a b s c",
  REPEAT GEN_TAC
  THEN BOOL_CASES_TAC "E:bool"
  THEN REWRITE_TAC [nAX_spec; nAXLemmaF; nAXLemmaT ];;
  nAX_correct =
  |- ! n E a b cin s c . nAX_imp n E a b cin s c = nAX_spec n E cin a b s c
  Run time: 0.3s
  Garbage collection time: 0.0s
  Intermediate theorems generated: 96
```

12.1.3 Verifying nAddXor

nAddXor is simply an nAX circuit with the carry-out pulled low. The specification and implementation definitions are (nAddXor was defined earlier):

```
#nAddXor_spec;;
|- ! n E a b cin s c .
  nAddXor_spec n E a b cin s c =
  (E => nAdder_spec n a b cin s c | (nXOR n a b cin s c /
    (c = F)))

#let nAddXor_imp = new_definition
  ('nAddXor_imp',
   "nAddXor_imp n E a b cin s c = ? q . (nAX_imp n E a b cin s q) /
    (and2_imp E q c ) " );;
  nAddXor_imp = ...
```

The verification is straightforward with only one interesting feature. We start by stripping away the quantifications and rewriting.

```
# "! E n a b cin s c .
  nAddXor_imp n E a b cin s c = nAddXor_spec n E a b cin s c"

 "! E n a b cin s c .
  nAddXor_imp n E a b cin s c = nAddXor_spec n E a b cin s c"

() : void
```
EXISTS_ELIM_TAC is not smart enough to cope with a relation for the existentially quantified variable that is not at the “top level”, and here it is embedded in a conditional. But by bool-casing on E, we can promote the relation to the top level in both sub-goals.

In the top sub-goal we have the relation \( c = q \). If we can turn this relation around, we can then use EXISTS_ELIM_TAC to substitute \( c \) for \( q \) throughout the left hand side. The fastest way of doing this is to use a conversion. Conversions and how terms are formed are treated in full in chapter 14; here we content ourselves by giving the intuition. We use the conversion \( \text{SYM} \text{CONV} \) to change the term \( c = q \) to \( q = c \), but first we have to navigate the conclusion of the goal and apply \( \text{SYM} \text{CONV} \) to the right sub-term. The steering consists of using \( \text{LHS} \text{CONV} \) to select the left hand side of the goal (which is a binder), using \( \text{BINDER} \text{CONV} \) to select the body of the binder (an application), and then \( \text{RAND} \text{CONV} \) to select the argument (the “rand”) of the body which is an application of \( \forall \) \( n\text{Adder}\_spec\ n\ a\ b\ s\ c\ q\ )\) to \( c = q \). Finally, conversions take terms to terms. \( \text{CONV_TAC} \) takes any conversion and transforms it into a tactic.
#e(let LHS_CONV = RATOR_CONV o RAND_CONV in
  let BINDER_CONV = RAND_CONV o ABS_CONV in
  CONV_TAC
  (LHS_CONV (BINDER_CONV (RAND_CONV SYM_CONV))));
OK..
"(\?q. nAdder_spec a b cin s q \land (q = c)) = nAdder_spec a b cin s c"
() : void

The rest of the proof is trivial.

#e(EXISTS_ELIM_TAC THEN REFL_TAC);
OK..
goal proved
|- (!E n a b cin s c.
  nAddXor_imp n E a b cin s c = nAddXor_spec n E a b cin s c)
Previous subproof:
"(\?q. (nXOR n a b s \land (q = a \land b n)) \land \neg c) = nXOR n a b s \land \neg c"
() : void

#e(EXISTS_ELIM_TAC THEN REFL_TAC);
OK..
goal proved
% <--> **** trace emitted **** => %
|- !E n a b cin s c.
  nAddXor_imp n E a b cin s c = nAddXor_spec n E a b cin s c
Previous subproof:
goal proved
() : void

Here is the proof in tidy form:
### 12.2 The nMod box

The nMod device is used to filter the a and b inputs to nAddXor according to the mode E and case selectors s0 and s1.

We specify and build a 1-bit Mod before specifying and verifying a row of them. The proof of the 1-bit Mod is uninteresting; that of the Mod sub-system is amenable to some structural twiddling and falls out easily if approached the right way. If one tries to bash it out by bool casing, the proof is easy but takes a lot of machine time due to the size of the goal.
12.2.1 Verification of the 1-bit Mod

Our specification of Mod has an obvious case structure in which we first split on mode (arithmetic or logical) and then do case analysis on the operations, each time giving a relation for $a'$ and $b'$.

```haskell
#let Mod_spec = new_definition
  ('Mod_spec',
   "Mod_spec E s0 s1 a b a' b' = (E => ( (~s0 \~ s1) => ((a' = a) \ (b' = F)) |
      (s0 \ s1) => ((a' = a) \ (b' = T)) |
      ( s0 \ ~s1) => ((a' = a) \ (b' = b)) |
      (a' = a) \ (b' = b))
    ) |
    \% not E \%
    ( (s0 \ ~s1) => ((a' = (a \/ b)) \ (b' = F)) |
      (s0 \ s1) => ((a' = a) \ (b' = T)) |
      ( s0 \ ~s1) => ((a' = a) \ (b' = b)) |
      (a' = (a \/ b)) \ (b' = b))
  )
) ;
Mod_spec = ...
```

The implementation follows Lewin (presumably because the full power of a mux is not needed to compute $a'$) in flattening the muxes. Although our model will not detect the difference, we have deviated from Lewin and followed normal design practice in re-computing values for $\overline{E}, \overline{s0}$ and $\overline{s1}$ internally at each stage rather than once for the whole device. We have also buffered values for $s0$ and $s1$. 
CHAPTER 12. STEP 3: ALU VERIFICATION

#let Mod_imp = new_definition
('Mod_imp',
  "Mod_imp E s0 s1 a b a' b' = ? bbar Ebar s0bar s1bar Eand Eor p q r s t u .
  (inv b bbar) /
  (inv E Ebar) /
  (inv s0 s0bar) /
  (inv s1 s1bar) /
  (and3_imp Ebar s0 s1 Eand) /
  (and3_imp Ebar s0bar s1bar Eor) /
  (and2_imp Eand b' t) /
  (and2_imp Eor b u) /
  (or3_imp a t u a') /
  (and3_imp s0bar s1bar F p) /
  (and3_imp s0bar s1 T q) /
  (and3_imp s0 s1bar b r) /
  (and3_imp s0 s1 bbar s) /
  (or4_imp p q r s b')
"));
Mod_imp = ...

The proof is trivial.

#let Mod_correct = prove_thm
('Mod_correct',
  " ! E s0 s1 a b a' b' .
  Mod_imp E s0 s1 a b a' b' = Mod_spec E s0 s1 a b a' b'",
  REPEAT GEN_TAC
  THEN PURE_REWRITE_TAC
  [ Mod_imp; inv;
    and3_correct; and3_spec;
    and2_correct; and2_spec;
    or4_correct; or4_spec;
    or3_correct; or3_spec
  ]
  THEN EXISTS_ELIM_TAC
  THEN REWRITE_TAC [ Mod_spec ]
  THEN MAP_EVERY BOOL_CASES_TAC
  [ "E:bool"; "s0:bool"; "s1:bool" ]
  THEN REWRITE_TAC []
);;
Mod_correct =
|- ! E s0 s1 a b a' b'.
  Mod_imp E s0 s1 a b a' b' = Mod_spec E s0 s1 a b a' b'
Run time: 30.3s
Garbage collection time: 11.0s
Intermediate theorems generated: 9197
12.2.2 Verification of nMod

In the specification of nMod

```hs
#let nMod_spec = new_definition
('nMod_spec',
  "nMod_spec n E s0 a b a' b'
  = (E => ( (s0 <= s1) => (nEql a' a n /\ nEql b' zeros n)
       | (s0 <= s1) => (nEql a' a n /\ nEql b' ones n)
       | (s0 <= s1) => (nEql a' a n /\ nEql b' b n)
       | (nEql a' a n /\ nEql b' (nNot b) n))
  | % not E => logical unit %
    ( (s0 <= s1)
      => (nEql a' (nOr a b) n /\ nEql b' zeros n)
      | (s0 <= s1)
      => (nEql a' a n /\ nEql b' ones n)
      | (s0 <= s1)
      => (nEql a' a n /\ nEql b' b n)
      | (nEql a' (nOr a (nNot b)) n /\ nEql b' (nNot b) n)))
)
"));
nMod_spec = ...
```

we have used nEql to give “finite” length to each relation for \(a'\) and \(b'\) in the specification. Informally, being primitive recursive, the implementation defines \(a'\) and \(b'\) for indices \(n+1\) down to 0, but says nothing about values of \(a'\) and \(b'\) beyond the size of the implementation. We have to restrict what the specification says about \(a'\) and \(b'\) to values at indices in the same range, otherwise we cannot get an equivalence proof. The \(\forall n \in \mathbb{N}, a' n = a n\) is too strong. We use nEql to put the appropriate size constraints on the specification relations, as in \(nEql a' a n\).

The implementation is defined as a regular structure by primitive recursion.

```hs
#let nMod_imp = new_prim_rec_definition
('nMod_imp',
  "nMod_imp 0 E s0 a b a' b'
  = Mod_imp E s0 a (a 0) (b 0) (a' 0) (b' 0))
\/
(nMod_imp (SUC n) E s0 a b a' b'
  = (nMod_imp n E s0 a b a' b') \/
    (Mod_imp E s0 a (a(SUC n)) (b(SUC n)) (a'(SUC n)) (b'(SUC n))))
"));
nMod_imp = ...
```

The proof is trivial but very time consuming if one resorts to bool-casing. Instead we use \(\pi\)IL to rearrange the conditional terms on the left
(arising from the rewritten implementation) which have the form
\[(b_1 \Rightarrow A_1 | b_2 \Rightarrow A_2 \ldots | A_n) \land (b_1 \Rightarrow a_1 | b_2 \Rightarrow a_2 \ldots | a_n)\] to the form
\[(b_1 \Rightarrow A_1 \land a_1 | b_2 \Rightarrow A_2 \land a_2 \ldots | A_n \land a_n)\] which has the same structure as the term on the right (arising from the specification).

We next juggle the sub-terms in each arm of the complicated conditional on the left so that they match their order on the right exactly.

All that is left is an unfolding of the definitions on the left and we are left with a tautology.

```
#let rIL = prove
  "\! a b c d .
    ((c \Rightarrow a \mid b) \land (c \Rightarrow d \mid e))
    = (c \Rightarrow (a \land d) \land (b \land e))",
  REPEAT GEN_TAC
  THEN BOOL_CASES_TAC "c:bool"
  THEN REWRITE_TAC [\[\[\]]];;
  rIL =
  |- \! a b c d . (c \Rightarrow a \mid b) \land (c \Rightarrow d \mid e) = (c \Rightarrow (a \land d) \land (b \land e))
```

```
#let rAL = prove
  "\! a b c d .
    (((a \land b) \land c) \land (b \land d))
    = (((a \land c) \land (b \land d))",
  REDUNDANT_GEN_TAC
  THEN PURE_REWRITE_TAC [CONJ_ASSOC]
  THEN CANCEL_CONJ_TAC;;
  rAL =
  |- \! a b c d . ((a \land b) \land c \land (b \land d) = (a \land c) \land (b \land d)
```

```
#let nMod_correct = prove_thm
  "nMod_correct",
  "\! n E s0 s a b a' b' .
    nMod_imp n E s0 s1 a b a' b' = nMod_spec n E s0 s1 a b a' b'",
  REDUNDANT_TAC THEN REPEAT GEN_TAC
  THEN PURE_ASM_REWRITE_TAC
  THEN PURE_ASM_REWRITE_TAC
  [ nMod_imp; nMod_spec; Mod_correct; Mod_spec ]
  THEN
  [ INDUCTION_TAC
    THEN PURE_ASM_REWRITE_TAC
    THEN PURE_ASM_REWRITE_TAC
    [ rIL ]
  ]
  THEN PURE_REWRITE_TAC
  [ nEq; bEq; zeros; ones; nOr; nNot ]
  THEN REFL_TAC
);;
```
12.3 ALU verification

In this section we complete the verification of the ALU. The proof is straightforward. We find that most of the pieces are in place and we need but two detours to prove helpful lemmata.

Here is our specification of the ALU.

```plaintext
#let nALU_spec = new_definition
(`nALU_spec',
 `nALU_spec n E s0 s1 cin a b s c =
  E => % arithmetic operations %
   ( (`s0 \ s1) => nINC n a cin s c 
    | (`s0 \ s1) => nDEC n a cin s c 
    | ( s0 \ s1) => nADD n a b cin s c 
    | nSUB n a b cin s c
   ) % logic operations %
   ( (`s0 \ s1) => (nOR n a b s \ (~c) / (c = F))
    | (`s0 \ s1) => (nNOT n a s \ (~c) / (c = F))
    | ( s0 \ s1) => (nXOR n a b s \ (~c) / (c = F))
    | (nAND n a b s \ (~c))
   ) ;;
)nALU_spec = ...;
```

and here are some “mini” tests of the specification which check to see that it behaves as expected when E is low

```plaintext
#REWRITE_RULE [] (SPECL ["n:num"; "F"] nALU_spec);;
|- !s0 s1 cin a b s c.
  nALU_spec n F s0 s1 cin a b s c =
  ((`s0 \ s1) =>
    (nOR n a b s \ (~c) | `(s0 \ s1) =>
    (nNOT n a s \ (~c) | `(s0 \ s1) =>
    (nXOR n a b s \ (~c) | (nAND n a b s \ (~c))));
```

and, when more refined, as an adder without carry

```plaintext
#REWRITE_RULE [] (SPECL ["n:num"; "T"; "F"; "F"] nALU_spec);;
|- !a b c. nALU_spec n T T F F a b s c = nADD n a F s c
```
or a bitwise exclusive or

```plaintext
#REWRITE_RULE [] (SPECL [ "n: num" ; "F" ; "T" ; "F" ; "F" ] nALU_spec);

\! a b s c . nALU_spec n F T F F a b s c = nxor n a b s /\ "c
```

The implementation bolts equally sized `nMod` and `nAddXor` boxes together.

```plaintext
#let nALU_imp = new_definition

('nALU_imp',
 "nALU_imp n E s0 s1 cin a b s c
 = \? a' b'.
   (nMod_imp n E s0 s1 a b a' b') /\
   (nAddXor_imp n E a' b' cin s c)
");;

nALU_imp = ...
```

We note that the specification of the ALU is textually very long. As ever the first few steps in the proof are concerned with tidying up the left hand side. It is clearer (and quicker to rewrite) if we refrain from expanding with `nALU_spec` for the time being.

```plaintext
#g " ! n E s0 s1 cin a b s c .
   nALU_imp n E s0 s1 cin a b s c
 = nALU_spec n E s0 s1 cin a b s c";;

"! n E s0 s1 cin a b s c .
   nALU_imp n E s0 s1 cin a b s c = nALU_spec n E s0 s1 cin a b s c"

() : void
```

```plaintext
#e(REPEAT GEN_TAC

THEOREM PURE_REWRITE_TAC

[ nALU_imp;
  nMod_correct; nMod_spec;
  nAddXor_correct; nAddXor_spec;
]
```

At this stage it is convenient to bool-case on $E$ and divide the proof into separate parts for the arithmetic operations and for the logical operations.
12.3.1 Verifying the arithmetic operations

We first probe the goal, splitting it into 4 cases, and see if any patterns emerge.

And indeed a pattern does emerge on the left. In each case we are given three conjuncts which: define values for a hidden bus \( a' \), define values for a hidden bus \( b' \), and then use them in \( \text{nAdder-spec} \). This suggests we prove a special lemma. We set a new goal on top of the stack and proceed with its proof.
DIVERSION I: nAdderLemma

```c

#g "! n f g cin s c .
    (a b .
       (nEq1 a f n ) /
       nAdderSpec n a b cin s c)
    = nAdderSpec n f g cin s c";;

"! n f g cin s c .
(a b . (nEq1 a f n ) /
    nAdderSpec n a b cin s c) =
    nAdderSpec n f g cin s c"

() : void


#e(REPEAT GEM_TAC
    THEN PURE_ONCE_REWRITE_TAC [nEq1Val; nAdder_spec]
    THEN let_TAC);;
OK..

"(! a b .
    (val a n = val f n ) /
    (val b n = val g n)) /
    (val s n =
     ((val a n ) + ((val b n ) + (bv cin))) < (2 EXP (SUC n )) =>
     (val a n ) + ((val b n ) + (bv cin)) |
     ((val a n ) + ((val b n ) + (bv cin))) - (2 EXP (SUC n ))) /
    (c = ((val a n ) + ((val b n ) + (bv cin))) < (2 EXP (SUC n )))) =
    (val s n =
     ((val f n ) + ((val g n ) + (bv cin))) < (2 EXP (SUC n )) =>
     (val f n ) + ((val g n ) + (bv cin)) |
     ((val f n ) + ((val g n ) + (bv cin))) - (2 EXP (SUC n ))) /
    (c = ((val f n ) + ((val g n ) + (bv cin))) < (2 EXP (SUC n )))"

() : void

```

Using EXISTSEP_ELIM_TAC is of no avail since the current version is not powerful enough to cope with relations like \( \exists a . \) val a n = expr. Instead we split the goal into two implications using EQ_TAC which turns a goal (asm, a => b) into two goals (asm, a => b) and (asm, b => a). We then call STRIP_TAC to push each antecedent onto its respective assumption list, after which we rewrite.
One subgoal has already been solved. The remaining subgoal is existentially quantified. We can remove hidden lines in goals of this form using \texttt{EXISTS\_TAC}, which strips away the leading existentially quantified variable and substitutes \texttt{term} for each free occurrence in the body. Thus applying \texttt{EXISTS\_TAC \ term} to a goal \texttt{(asm, \ ? a . body)} results in a new goal \texttt{(asm, \ [term/a]body)}.

In this case, we chose two obvious substitutions that trivialised the first two conjuncts. \textbf{N.B.} Notice that if we select inappropriate arguments to
EXISTS_TAC we can render a provable goal unprovable, but we cannot render an unprovable goal provable. Soundness is preserved.

There is no need to rewrite this goal. It is the conjunction of three terms each of which is a reflection. We split the goal into separate subgoals for each conjunct and then apply REFL_TAC.

```ocaml
# e (REPEAT CONJ_TAC THEN REFL_TAC);;
ok

goal proved

% << ***** trace omitted ***** >> %

|- !n f g cin s c.
   (?a b. (nEqz a f n \ nEqz b g n) \ nAdder_spec n a b cin s c) = nAdder_spec n f g cin s c

Previous subproof:
goal proved

() : void
```

Here is the tidied proof.

```ocaml
# let nAdderLemma = prove

("! n f g cin s c.
   (? a b.
      (nEqz a f n \ nEqz b g n) \ nAdder_spec n a b cin s c)
   = nAdder_spec n f g cin s c",
   REPEAT GEN_TAC
   THEN PURE_ONCE_REWRITE_TAC [ nEqzVal; nAdder_spec ]
   THEN let_TAC
   THEN EQ_TAC THEN STRIP_TAC
   THEN ASM_REWRITE_TAC []
   THEN MAP_EVERY EXISTS_TAC [ "f:num\rightarrow bool"; "g:num\rightarrow bool" ]
   THEN REPEAT CONJ_TAC THEN REFL_TAC);;
nAdderLemma =

|- !n f g cin s c.
   (? a b. (nEqz a f n \ nEqz b g n) \ nAdder_spec n a b cin s c) = nAdder_spec n f g cin s c

Run time: 1.5s
Intermediate theorems generated: 222
```
... and back to the main proof

We backup past the remnants of the nAdderLemma proof to where we were, go back one more stage (past the probe), then bool-case and rewrite with nAdderLemma.

\[
\text{#b();b();b();b();b();b();}\\%
\text{<< ***** backup omitted ***** >> %}
\text{"(a\ b).}
\text{((s0 \| s1) =>}
\text{(nEql a' a n \& nEql b' zeros n) |}
\text{((s0 \| s1) =>}
\text{(nEql a' a n \& nEql b' ones n) |}
\text{((s0 \| s1) =>}
\text{(nEql a' a n \& nEql b' b n) |}
\text{((s0 \| s1) =>}
\text{nAdder.spec a' a' b' cin s c) =}
\text{nALU.spec n T s0 s1 cin a b s c"}
\text{)} : void}
\]

We are nearly there. Exercise 11.6 asked you to prove some simple lemmata, each of which takes care of one of these sub-goals. For example,

\[
\text{#nALUSub;}\\%
\text{! n a b cin s c.}
\text{nALU.spec n T T T cin a b s c = nAdder.spec n a(nNot b)cin s c}
\]

We take these theorems and pass them as arguments to ACCEPT_TAC o SYM_RULE o SPEC_ALL .
We omit the trace of the next three cases which follow the same pattern.

### 12.3.2 Verifying the logical operations

We are now left with the verification of the logical operations. Here is the subgoal.

Once more it seems sensible to probe the goal by bool-casing
and again a common pattern emerges on the left. We take time out to prove an appropriate lemma which will enable us to clear away the hidden lines in all four subgoals.

### Diversion II: nXorlemma

We set the goal to be proved:

```ocaml
# e (MAP EVERY BOOL CASES_TAC ['"s0:bool"'; '"s1:bool"'])
THEN REWRITE_TAC [];;
```

```
OK.
4 msubgoals
"(7a' b'.
   (nEqI a'(nOr a b) n /
    nEqI b' zeros n) /
    nXOr n a' b' s /
    '~c) =
   nALU_spec n F F cin a b s c"

"(7a' b'. (nEqI a' a n /
    nEqI b' ones n) /
    nXOr n a' b' s /
    '~c) =
   nALU_spec n F T cin a b s c"

"(7a' b'. (nEqI a' a n /
    nEqI b' b n) /
    nXOr n a' b' s /
    '~c) =
   nALU_spec n F T cin a b s c"

"(7a' b'.
   (nEqI a'(nOr a(nNot b)) n /
    nEqI b' (nNot b) n) /
    nXOr n a' b' s /
    '~c) =
   nALU_spec n F T cin a b s c"

() : void
```

Since we have to be able to expand the definition of nEqI, we start the proof by inducting on n.
Question: why did we rewrite with \texttt{nXOR} before inducting?

\textbf{Base case.} Trivial.

\begin{verbatim}
#e(PURE_ONCE_REWRITE_TAC [ nXOR ] )
  THEN INDUCT_TAC THEN REPEAT GEN_TAC
  THEN PURE_REWRITE_TAC [ nEq; bvEq; nXor ];
OK..
2 subgoals
  "(?a b.
    ((nEq1 a f n \ (a(SUC n) = f(SUC n))) /\  nEq1 b g n /\  (b(SUC n) = g(SUC n))) /\  (nEq1 s(nXor a b)n \ (s(SUC n) = "a(SUC n) = b(SUC n))) /\  ~c) =
    (nEq1 s(nXor f g)n \ (s(SUC n) = "f(SUC n) = g(SUC n))) /\  ~c"
  [ "f g = c.  
    (?a b. (nEq a f n \ nEq b g n) \ nEq s(nXor a b)n \ ~c) =
    nEq s(nXor f g)n /\  ~c"
  ]

  "(?a b. (a 0 = f 0) \ (b 0 = g 0)) /\ (s 0 = "(a 0 = b 0)) /\  ~c) =
    (s 0 = "(f 0 = g 0)) /\  ~c"
  () : void
\end{verbatim}

\begin{verbatim}
#e(EQ_TAC THEN STRIP_TAC
  THEN ASM_REWRITE_TAC []);
OK.
"?a b. (a 0 = f 0) \ (b 0 = g 0)) /\ (s 0 = "(a 0 = b 0))"
  [ "s 0 = "(f 0 = g 0)"
  [  "c"
  ]
  ( ) : void
\end{verbatim}

\begin{verbatim}
#e(MAP_EVERY EXISTS_TAC [ "f:num->bool", "g:num->bool" ]);;
OK.
"((f 0 = f 0) /\ (g 0 = g 0)) /\ (s 0 = "(f 0 = g 0))"
  [ "s 0 = "(f 0 = g 0)"
  [  "c"
  ]
  ( ) : void
\end{verbatim}
CHAPTER 12. STEP 3: ALU VERIFICATION

The induction step involves some interesting and delicate rewriting from the assumption list. Since EXISTS_ELIM_TAC does nothing unless the goal contains relations for the quantified variables, we start by splitting the goal into two implications and pushing the antecedents onto their respective assumption lists.
12.3. ALU VERIFICATION

"(n≡l s (nXor f g)n /\ (s(SUC n) = ~(f(SUC n) = g(SUC n)))) /\ ~c"

[] "f g = c.
( n a b. (n≡l a f n /\ n≡l b g n) /\ n≡l s (nXor a b)n /\ ~c) =
 n≡l s (nXor f g)n /\ ~c"

[ "n≡l a f n" ]
[ "a(SUC n) = f(SUC n)"
[ "n≡l b g n" ]
[ "b(SUC n) = g(SUC n)"
[ "n≡l s (nXor a b)n" ]
[ "s(SUC n) = ~(a(SUC n) = b(SUC n))"
[ ~c"

() : void

A quick rewrite from the assumptions makes a lot of simplifications.

#e(ASM_REWRITE_TAC []);
OK...
"n≡l s (nXor f g)n"

[] "f g = c.
( n a b. (n≡l a f n /\ n≡l b g n) /\ n≡l s (nXor a b)n /\ ~c) =
 n≡l s (nXor f g)n /\ ~c"

[ "n≡l a f n" ]
[ "a(SUC n) = f(SUC n)"
[ "n≡l b g n" ]
[ "b(SUC n) = g(SUC n)"
[ "n≡l s (nXor a b)n" ]
[ "s(SUC n) = ~(a(SUC n) = b(SUC n))"
[ ~c"

() : void

Too many in fact since we are left with a goal that doesn’t match with anything on the assumption list, although we could conjoin the goal with ~c which is on the assumption list. Instead we backup and try a more delicate approach to rewriting from the assumptions.

#b();

% << **** trace emitted **** >> %

We first remove the middle conjunct by rewriting with just the assumptions "a(SUC n) = f(SUC n)", "b(SUC n) = g(SUC n)", and "s(SUC n) = ~(a(SUC n) = b(SUC n))". These can be selected from the assumption list by a judicious use of FILTER_ASM_REWRITE_TAC. Given a goal (asm, c), FILTER_ASM_REWRITE_TAC p L works as follows. It calls filter p asm and accepts only those assumptions which pass through the filter (is_sj eq in our
case). Rewriting is then carried out with the filtered assumptions and with the user-supplied list of rewrites L.

```plaintext
#e(FILTER_ASM_REWRITE_TAC is_eq []);
OK.
"nEq1 s(nXor f g)n \land \neg c"
[ "!f g s c.
  (?a b. (nEq1 a f n \land nEq1 b g n) \land nEq1 s(nXor a b)n \land \neg c) =
  nEq1 s(nXor f g)n \land \neg c"
]
[ "nEq1 a f n"
]
[ "a(SUC n) = f(SUC n)"
]
[ "nEq1 b g n"
]
[ "b(SUC n) = g(SUC n)"
]
[ "nEq1 s(nXor a b)n"
]
[ "s(SUC n) = \neg (a(SUC n) = b(SUC n))"
]
[ "\neg c"
]
() : void
```

The new goal appears in the assumption list on one right hand side. We now wish to pick out just that assumption, turn it around (using SYM_RULE), and then rewrite with it. We introduce ASSUM_LIST, the HOL primitive for accessing all the assumptions. ASSUM_LIST tac makes a copy of the assumption list by assuming each term on it in turn, and passes this as the parameter to tac. Notice that FILTER_ASM_REWRITE_TAC is defined in terms of ASSUM_LIST by:

```plaintext
#ASSUM_LIST;;
- : ((thm list -> tactic) -> tactic)

#let FILTER_ASM_REWRITE_TAC p L
  = ASSUM_LIST
    (\asl. REWRITE_TAC ((filter (p o concl) asl) @ L));
FILTER_ASM_REWRITE_TAC = - : ((term -> bool) -> thm list -> tactic)
```

Here is an appropriate call and a trace of it in action. We use (asm, C) as a short hand for the goal.

```plaintext
(ASM_LIST ( PURE_ONCE_REWRITE_TAC
  o (map SYM_RULE)
  o (filter (is_forall o concl))
 ))
--- ASSUM_LIST ( PURE_ONCE_REWRITE_TAC
  o (map SYM_RULE)
  o (filter (is_forall o concl))
 ) (asm, C)
--- PURE_ONCE_REWRITE_TAC
  (map SYM_RULE
```

ASSUM_LIST takes the assumption list (a list of terms) and maps ASSUME down it, it hands over a list of theorems to its tactic argument. Hence when we filter, we have to test its conclusion for being a quantified term.

This leg of the proof closes by our removing the quantifiers and then rewriting with other assumptions.
CHAPTER 12. STEP 3: ALU VERIFICATION

#e MAP_EVERY_EXISTS_TAC [ "a: num->bool" ; "b: num->bool" ]
THEN A SM_REWRITE_TAC [] ;
OK .
goal proved
.... |- ?a b. (nEq1 a f n /
       nEq1 b g n) /
       nEq1 s(nXor a b)n /
       "c
.... |- nEq1 s(nXor f g)n /
     "c
........ |- nEq1 s(nXor f g)n /
       (s(SUC n) = "(f(SUC n) = g(SUC n))) /
       "c

Previous subproof:
"?a b.
((nEq1 a f n /
 (a(SUC n) = f(SUC n))) /
 nEq1 b g n /
 (b(SUC n) = g(SUC n))) /
 (nEq1 s(nXor a b)n /
 (s(SUC n) = "(a(SUC n) = b(SUC n))) /
 "c"

[ "!f g s c.
 (?a b. (nEq1 a f n /
 nEq1 b g n) /
 nEq1 s(nXor a b)n /
 "c) =
 nEq1 s(nXor f g)n /
 "c"
]
[ "nEq1 s(nXor f g)n" ]
[ "s(SUC n) = "(f(SUC n) = g(SUC n))"
]
[ ""c" ]

() : void

The last steps are obvious.

#e MAP_EVERY_EXISTS_TAC [ "f: num->bool" ; "g: num->bool" ]
THEN A SM_REWRITE_TAC [] ;
OK .
"nEq1 f f n /
 nEq1 g g n"

[ "!f g s c.
 (?a b. (nEq1 a f n /
 nEq1 b g n) /
 nEq1 s(nXor a b)n /
 "c) =
 nEq1 s(nXor f g)n /
 "c"
]
[ "nEq1 s(nXor f g)n" ]
[ "s(SUC n) = "(f(SUC n) = g(SUC n))"
]
[ ""c" ]

() : void

#e MAP_EVERY_EXISTS_TAC ["nEq1Val;]
|- !n a b. nEq1 a b n = (val a n = val b n)

#e REWRITE_TAC [ nEq1Val ] ;
OK .
goal proved

% << ***** trace omitted ***** >> %
12.3. ALU VERIFICATION

\[ \neg \neg f g \neg c. \]
\[ (\exists a b. (nEql a f n \land nEql b g n) \land \neg XOR n a b s \land \neg c) =\]
\[ nXOR n f g s \land \neg c \]

Previous subproof:
goal proved
() : void

Here is the tidied lemma.

```plaintext
#let nXorLemma = prove
("\neg \neg f g \neg c.
(\exists a b. (nEql a f n \land nEql b g n) \land (nXOR n a b s \land \neg c) =
(nXOR n f g s) \land \neg c)",&
PURE_ONCE_REWRITE_TAC [ nXOR ]
THEN INDUCT_TAC THEN REPEAT GEN_TAC
THEN PURE_REWRITE_TAC [ nEql; bvEql; nXor ]
THEN,
[ EQ_TAC THEN STRIP_TAC
THEN ASM_REWRITE_TAC []
THEN MAP_EVERY EXISTS_TAC
[ "f:num->bool"; "g:num->bool" ]
THEN REWRITE_TAC []]
;
EQ_TAC THEN STRIP_TAC
THEN,
[ FILTER_ASM_REWRITE_TAC is_eq []
THEN ASSUM_LIST
( PURE_ONCE_REWRITE_TAC
  o (map GSYM)
  o (filter (is_forall o concl))
)
THEN MAP_EVERY EXISTS_TAC
[ "a:num->bool"; "b:num->bool" ]
THEN ASM_REWRITE_TAC []
;
MAP_EVERY EXISTS_TAC
[ "f:num->bool"; "g:num->bool" ]
THEN ASM_REWRITE_TAC [ nEqlVal ]
]
)
)
);;
nXorLemma =
\[ \neg \neg f g \neg c.
(\exists a b. (nEql a f n \land nEql b g n) \land \neg XOR n a b s \land \neg c) =\]
\[ nXOR n f g s \land \neg c \]
Run time: 2.3s
Garbage collection time: 1.0s
Intermediate theorems generated: 513
```
... and back to the main proof again

We back up past the above proof to the main proof

```
#b();b();b();b();b();b();b();b();b();b();b();b();

% << ***** trace omitted ***** >> %

"(??a' b'.
 ("s0 /
 (nEq1 a' (nOr a b) n /
 nEq1 b' zeros n) |
 ("s0 /
 (nEq1 a' a n /
 nEq1 b' ones n) |
 ("s0 /
 (nEq1 a' a n /
 nEq1 b' b n) |
 (nEq1 a' (nOr a (nNot b)) n /
 nEq1 b' (nNot b)n))) /
 nXOr n a' b' s /

"c) =
 nALU_spec n F s0 s1 cin a b s c"

() : void
```

and then bool-case and rewrite with nXorlemma.

```
#(MAP_EVERY BOOL_CASES_TAC [ "s0:bool"; "s1:bool" ]
 THEN Rewrite_TAC [ nXorlemma ]);;
OK . .
4 subgoals

"nXOr n (nOr a b) zeros s /

"c = nALU_spec n F F F cin a b s c"

"nXOr n a ones s /

"c = nALU_spec n F F T cin a b s c"

"nXOr n a b s /

"c = nALU_spec n F T F cin a b s c"

"nXOr n (nOr a (nNot b)) (nNot b)s /

"c = nALU_spec n F T T cin a b s c"

() : void
```

Here is the next step which has been rendered trivial by exercise 11.7.

```
#nALUAnd;

[- !n a b s c.

nALU_spec n F T T cin a b s c = nXOr n (nOr a (nNot b)) (nNot b)s /

"c
```
The last three steps are similar. The complete proof in tidy form reads:

```plaintext
#let nALU_correct = prove_thm
('nALU_correct'),

"! n E s0 s1 cin a b s c .
   nALU_imp n E s0 s1 cin a b s c
   = nALU_spec n E s0 s1 cin a b s c",
REPEAT GEN_TAC
THEN PURE_REWRITE_TAC
[ nALU_imp;
  nMod_correct; nMod_spec;
  nAddXor_correct; nAddXor_spec
]
THEN MAP_EVERY BOOL_CASES_TAC
[ "E:bool"; "s0:bool"; "s1:bool" ]
THEN REWRITE_TAC [ nAdderLemma; nXorLemma ]
THEML (map (ACCEPT_TAC o SYM_RULE o SPEC_ALL)
[ nALUSub; nALUAdd; nALUDec; nALUInc;
  nALUAnd; nALUXor; nALUNot; nALUOr ]));

nALU_correct = |
  ! n E s0 s1 cin a b s c .
  nALU_imp n E s0 s1 cin a b s c = nALU_spec n E s0 s1 cin a b s c
Run time: 4.2s
Garbage collection time: 1.8s
Intermediate theorems generated: 701
```
EXERCISES 12

Exercise 12.1 Specify and verify the ALU designed and verified in [2, 14].
Part V

Filling the HOLes
We have now reached a reasonable level in our understanding of HOL. Because we developed and motivated HOL via examples, there is some unevenness in our knowledge of HOL methods and tools. However the examples were chosen so that we have at least seen and used the main weapons in the HOL arsenal and so we have at least a flavour of the whole picture. Now is the time to round this out. In part V we devote separate chapters to substitution, conversion and rewriting, tactics and tacticals, and theorem continuations. We survey and catalogue the most common of these tools and their uses, and explain how they fit together.

As with _HOL_, the major datatypes in HOL are _term_ and _thm_. Each theorem is held as a sequent—a pair of type _term list ≠ term_ where the _term list_ is interpreted as a list of hypotheses (assumptions) and the _term_ states the conclusion that holds under these assumptions.

It is important to grasp the distinction between substitution and rewriting. _Substitution_ can be used to replace one, selected or all occurrences of an “old” subterm by a “new” subterm in a third term. Notice that we can always replace one subterm by another in a third term—all we wind up with is another term, and there is no necessity for a term to be always true. Notice also that substitution _never_ uses pattern matching. It is up to the user to get the old and the new subterms in exactly the right form. Whereas substitution is defined as an operation on terms, _rewriting_ is an operation which turns one theorem into another theorem. There are two major differences between substitution and rewriting. (i) We cannot blindly replace occurrences of _expr_1 in a theorem by _expr_2 and expect to get another theorem. The operation of replacing like by like in a theorem is only valid if we have a proof that they are indeed equivalent, that is, we have a theorem |- _expr_1 = _expr_2. (ii) Pattern matching is built into rewriting. Since theorems are held as sequents, the rewriting tools do, of course, make use of the substitution primitives.

We start chapter 13 by explaining how terms are represented and how they may be constructed and taken apart. Then we define the mathematics of _substitution_ on terms and present the relevant built-in HOL primitives. We next cover _conversions_, which take terms to theorems. For example, _num_CONV "2"_ returns |- 2 = Suc 1. In general, a conversion takes a term,
say \( t_1 \), formally converts it to an equivalent term, say \( t_2 \), and returns a theorem \( t_1 = t_2 \). The theorem may then be used as an argument to a rewriting rule or a rewriting tactic. Even better, any conversion can be turned into an inference rule or into a tactic. \texttt{CONV_RULE conv} turns a conversion \texttt{conv} into an inference rule. \texttt{CONV_TAC conv} turns a conversion \texttt{conv} into a tactic. Writing the conversion first makes sure that the dependent rule and tactic work in a consistent and predictable manner. A conversion only applies to a term at the top level. Some control structure is required to navigate through a term and apply a conversion to all or selected subterms, once or repeatedly. HOL has several defined primitives which can be combined to produce these effects. Analogous to tacticals, they are called conversionals. In the final section of this chapter, we survey and catalogue different styles of forward inferencing.

In chapter 14 we examine tactics and tacticals. Because the proofs we carry out are large, we usually perform a proof in the backwards direction. We first set up a goal (a sequent, but this time the term is interpreted as the theorem we wish to prove), and then repeatedly use either rewriting to simplify the current goal, or apply goal decomposition tactics (e.g. \texttt{INDUCT_TAC} or \texttt{BOOL_CASES_TAC}) which split the goal into subgoals, until we are left with something that is easily solvable. Some of these “easy” goals are so trivial they can be solved at once, e.g. by using \texttt{REFL_TAC} or \texttt{ACCEPT_TAC} or by rewriting from the assumptions. But sometimes the built-in theorems don’t have quite the right “shape”. Then we use forward inferencing (with, for example, \texttt{MATCH_MP}) to massage existing theorems into the right shape.

In large and complex HOL proofs, it is quite usual for us to build up a number of terms on the assumption list. Often these assumptions are not quite what we want but are put on the list for further manipulation (for example, by \texttt{IMP_RES_TAC} or \texttt{RES_TAC}). A common side effect is that several other theorems that we never need also get added to the assumption list. Again we frequently use an assumption on the assumption list once but leave it there until the proof is completed. Thus there is a tendency for assumption lists to accumulate assumptions that we do not need any more. Is there anything we can do about this? The obvious way is intercept and manipulate assumptions and use them without adding them to the assumption list at all, or if that is unwieldy, then at least to manipulate them into a better shape before putting them onto the assumption list. In chapter 15, we categorise and catalogue various useful techniques and give some simple examples.

Much of these material is gleaned from Larry Paulson’s excellent text [90] and papers [88], from Mike Gordon’s overview of the HOL system [44], and from the HOL reference manual, system description and tutorial
Since this is an introductory text we do not explain everything in complete detail. Whilst we hope to tell you the truth and nothing but the truth, we haven’t the space to tell you the whole truth. In case of doubt, [106] is the final arbiter.

13.1 Representing HOL terms

The HOL system allows the user to submit and receive terms and theorems in a high level surface notation (double quoted). Once parsed, terms are held as ML data structures. Remember that HOL is intended as an extendible kernel not as a complete system, and that you are expected to tailor your own conversions and tactics. In order to do this you must be able to recognize, pick apart, and construct HOL terms and theorems, so you need a thorough understanding of the underlying ML implementation.

The abstract data type for terms is very compact with only 4 categories of term: constant, variable, application, and abstraction. Non-primitive terms are built on top of these 4 categories in the way we developed the pHOL system in chapter 3.

As we would predict from our experiences with pHOL, writing terms directly in ML is rather cluttered and so HOL is provided with a parser which transforms the HOL source in double quotes into the appropriate abstract data type representation. For example, "x \ F" is translated to

\[\text{mk_conj}(\text{mk_var}(\text{`x'}, \ "\text{bool}\"), \text{mk_const}(\text{`F'}, \ "\text{bool}\"))\]

—correctly, so it would seem

```
#(mk_conj(mk_var('x', "bool"), mk_const('F', "bool"))) = "x \ F";;
ttrue : bool
```

13.1.1 Primitive HOL terms

Here is a segment of the abstract data type for terms. Taken from left to right the constructors are: constant, variable, application, and abstraction.

```
abstract type term
  = string # type | string # type | term # term | term # term
with
  .......
```

The body of the abstract data type contains the functions which enable the construction, destruction, and querying of primitive terms. We enumerate these functions and show simple examples of their use.

**Constructing primitive terms.** Four primitive constructor functions are provided.

```ml
#(mk_const, mk_var, mk_comb, mk_abs);;
((-, ()), (-), (-), -)
: (((string # type) -> term) #
  (((string # type) -> term) #
  (((term # term) -> term) #
  (((term # term) -> term)))
```

The representations of constants and variables contain their types. The types of applications and abstractions are inferred from those of their constituents. We now construct some primitive terms.

```ml
#let x = mk_var ('x', "bool")
    and y = mk_var ('y', "bool")
    and z = mk_const('F', "bool");;
x = "x" : term
y = "y" : term
z = "F" : term
```

The use of `mk_const` is restricted to known constants; you cannot create a new constant without its first being declared. Furthermore, the type given for the constant must match its previously declared type, or be an instance of a variable type. Properties such as infix status are attached to the constant at the time of declaration, and the constants created using `mk_const` inherit this status.

Constants are usually declared when they are defined, as occurred when we used `new.definition`. We can give binary operators infix status by using `new.infix.definition` instead. For example, here is a definition of an implication operator:

```ml
#let implies = new_infix_definition
    ('implies',
     "(implies (x:bool) (y:bool)) = (\x y x implies y = "x \/
      y
#"(x implies y) /\ (y implies x) = (x = y);;
"x implies y /\ y implies x = (x = y) : term
```

The definition of `implies` is done in prefix style, but once defined it is used as an infix as shown. Should you wish to use an infix operator in prefix style, all you need do is prefix its identifier with $\$, as in `$\text{implies}$` below.
Many of the built-in binary operators are defined as infixes (e.g. +, -, *, DIV, MOD, EXP, =, <, >, <>, >, ....)

```haskell
let boolq = mk_const('implies', "bool->bool->bool");
boolq = "implies" : term

let ifThen = mk_const('==>', "bool->bool->bool");
ifThen = "==>": term

let c = mk_comb(mk_comb(ifThen, x), y);
c = "x ==> y" : term

let a = mk_abs(x, (mk_abs(y, c)));
a = "\(\lambda x. y. x \Rightarrow y\)" : term

mk_comb(a, z);
"(\(\lambda y. x \Rightarrow y\)F)" : term
```

We require two applications to represent "x ==> y". Notice the way the implication operator is printed; "==>": indicating that "==>": is prefix.

**Taking primitive terms apart.** Four destructor functions are provided for taking terms apart. The function `rator` returns the operator (function) of an application; the function `rand` returns the operand (argument) of an application. All calls on these functions will fail if the argument is not of the expected type.

```haskell
(dest_const, dest_var, dest_comb, dest_abs);
((-), (-), (-), -)
: (term -> (string # type)) #
(term -> (string # type)) #
(term -> (term # term)) #
(term -> (term # term))

(dest_const ifThen, dest_var x);
(==>, "bool->(bool->bool)", 'x', ":bool")
: ((string # type) # string # type)

(dest_comb c);
("==> x", "y") : (term # term)

(rator c, rand c);
("==> x", "y") : (term # term)
```
CHAPTER 13. SUBST, CONV, AND REWRITING

Querying primitive terms. Four primitive recognizer functions are provided.

13.1.2 Non-primitive HOL terms

Constructing and taking apart HOL terms would be very tedious if we kept to the primitives outlined above. Another layer of functionality has been added in HOL. For each and every one of the stems

```
#(is_const, is_var);
((true, false) : (bool -> bool) # (bool -> bool))
```

there are `mk`, `dest`, `is`-prefixes which may be used to generate, take apart, or query (respectively) a term of the type indicated by the stem.

These “higher level” functions are all built upon the primitives described in the last section. Together with their type information, they are tabulated in appendix A.

We illustrate the idea of making and taking apart higher level structures with the stem `imp`. To define `mk_imp` we first define the generic `mk_bin`. 
We remind you that the term we have constructed is the same as

---

We can define dest_imp in a similar vein.

---

We leave it to you to write is_imp.

Example 13.1.1 Construct the term "! x . ? y . y = x + 1".
CHAPTER 13. SUBST, CONV, AND REWRITING

Example 13.1.2 Lines (definition)

The second example is taken from [44, page 122]. The function `lines` will break apart a term of the form `! t . x t = rhs` and return `true` iff the name of the signal on the left (`x`) is a member of a certain list. A call `lines NAMES TM` returns false if `TM` is “not of the right shape” or if the variable on the left hand side is not a member of the name list `NAMES`.

```ocaml
#let lines names tm
    = ( let x = ( fst o dest_var
                   o fst o dest_comb
                   o fst o dest_eq
                   o snd o dest_forall
                    ) tm in
         mem x names ) ? false;;
lines = - : (string list -> term -> bool)
#lines ['y'; 'x'] "!t. (x:num->bool) t = rhs";;
true : bool
```

Since we only use one of the arguments as we move from one `let` to the next, we could also code `lines` by

```ocaml
#let lines names tm
    = ( let x = ( fst o dest_var
                   o fst o dest_comb
                   o fst o dest_eq
                   o snd o dest_forall
                    ) tm in
         mem x names ) ? false;;
lines = - : (string list -> term -> bool)
```

but we think that the first style is a little easier to read, write and maintain.

13.1.3 Substitution in terms

Roughly speaking, substitution takes two terms, say \( \alpha \) and \( \beta \), and a term, say \( tm \), and replaces all free occurrences of the term \( \alpha \) in \( tm \) by the term \( \beta \).
Importantly, no pattern matching takes place. The substitution primitives are heavily used primitives in rewriting agents.

In general we can substitute any term for any other term, provided that they are of the same type. Indeed, substitution is not carried out unless the types agree. Since substitution is notoriously difficult to get right\(^1\), we start by considering substitution for free variables and then infer the general case of substitution for free terms.

**Free and bound variables**

We need the definitions of *bound* and *free* variables before we can give the algorithm for substitution. A variable \( x \) is bound only when it occurs in the body of an abstraction whose bound variable is \( x \); otherwise it is free. The detailed definitions are given by induction over the structure of HOL terms: constants, variables, applications and abstractions. NB. In the following, \( x \equiv y \) means \( x \) and \( y \) are the same variable; \( x \not\equiv y \) means \( x \) and \( y \) are different variables.

\textbf{isFree} \( x \ M \) — is the variable \( x \) free in the term \( M \)?

- \( M \) is a constant.
  - \( x \) is not free in \( M \).

- \( M \) is the variable \( y \).
  - \( x \) is free in \( M \) iff \( x \equiv y \).

- \( M \) is the application \( B \ C \).
  - \( x \) is free in \( M \) if \( x \) is free in \( B \), or in \( C \), or in both.

- \( M \) is the abstraction \( \backslash y \ . \ . B \).
  - If \( x \equiv y \), then \( x \) is not free in \( M \).
  - If \( x \not\equiv y \), then \( x \) is free in \( M \) iff \( x \) is free in \( B \).

\textbf{isBound} \( x \ M \) — is the variable \( x \) bound in the term \( M \)? The only way to bind a variable in a primitive HOL term is through an abstraction.

- \( M \) is a constant.
  - \( x \) is not bound in \( M \).

- \( M \) is a variable.
  - \( x \) is not bound in \( M \).

\(^1\) According to [32]: “Most formulations of the rule for substitution which were published even by the ablest logicians, before 1546, were demonstrably incorrect.”
• \( M \) is the application \( BC \).
  \( x \) is bound in \( M \) iff \( x \) is bound in \( B \), or in \( C \), or in both.

• \( M \) is the abstraction \( \lambda y.B \).
  If \( x \equiv y \), then \( x \) is bound in \( M \) iff it occurs in \( B \).
  If \( x \not\equiv y \), then \( x \) is bound in \( M \) iff \( x \) is bound in \( B \).

\( \text{Note that it is possible for a variable to be both free and bound in the same term, witness } x \text{ in } (\lambda x . x) (x + y) \text{ where } x \text{ is bound in the rator } (\lambda x . x) \text{ and free in the rand } (x + y). \)

**Substitution for free variables**

For any terms \( M \) and \( N \) and variable \( x \), where \( N \) and \( x \) have the same type, we define \( [N/x]M \) to be the result of substituting \( N \) for every free occurrence of \( x \) in \( M \), changing bound variables to avoid name clashes where appropriate.

\( [N/x]M \) — the result of substituting \( N \) for each free \( x \) in \( M \) (where \( x \) of the same type as \( N \)) — is defined by induction on \( M \).

• \( M \) is the constant \( c \).
  \( [N/x]c = c \)

• \( M \) is a variable.
  \( [N/x]x = N \).
  If \( x \not\equiv y \), then \( [N/x]y = y \)

• \( M \) is the application \( BC \).
  \( [N/x](B C) = ([N/x]B)([N/x]C) \)

• \( M \) is an abstraction.
  \( [N/x](\lambda x.P) = (\lambda x.[N/x]P) \)
  If \( y \not\equiv x \), then \( [N/x](\lambda y.P) = \lambda y. [N/x]P \), if \( y \) is not free in \( N \)
  = \( \lambda y . P \), if \( x \) is not free in \( P \)
  = \( \lambda z . [N/x]([z/y]P) \), if \( y \) is free in \( N \) and \( x \) is free in \( P \) and \( z \) is an introduced variable which is not free in either \( N \) or in \( P \).
13.1.4 Substitution for free terms

Slind [102] has generalised our definition of substitution to allow the $x$ in $[N/x]M$ to be a free term. Intuitively a term $tm$ is free in another term $M$ if there is an occurrence of $tm$ in $M$ in which none of the free variables of $tm$ are bound within $M$. Thus $f\ 0$ is free in $f\ (f\ 0)$ but not in $\ f\ .\ f\ 0$.

**isFreeInTerm tm M**—is the term $tm$ free in the term $M$?
The definition is by cases, then by induction on $M$. $tm \equiv M$ may be read as “$tm$ is identical to $M$”.

- $tm \equiv M$.
  $tm$ is free in $M$.

- $tm \not\equiv M$.
  - $M$ is the application $B\ C$.
    - $tm$ is free in $M$ if $tm$ is free in $B$, or in $C$, or in both.
  - $M$ is the abstraction $\ \backslash \ x \ .\ B$.
    - If $x$ is free in $tm$, then $tm$ is not free in $M$.
    - If $x$ is not free in $tm$, then $tm$ is free in $M$ iff $tm$ is free in $B$.
  - Otherwise.
    - $tm$ is not free in $M$.

$[N/tm]M$—the result of substituting $N$ for each free $tm$ in $M$ (where $tm$ of the same type as $N$) is defined by cases and then by induction on $M$.

- $tm \equiv M$.
  $tm = N$

- $tm \not\equiv M$.
  - $M$ is a constant $c$.
    - $[N/tm]c = c$.
  - $M$ is a variable $v$.
    - $[N/tm]v = v$.
  - $M$ is the application $B\ C$.
    - $[N/tm](B\ C) = (\ [N/tm]B)\ ([N/tm]C)$
  - $M$ is an abstraction.
    - $[N/tm](\ \backslash \ x \ .\ P)$
      $= (\ \backslash \ x \ .\ P)\ if\ x\ is\ free\ in\ tm\ or\ if\ tm\ is\ not\ free\ in\ P$. 
\[ = (\forall x \cdot [\text{tm}/P]\text{) if } x \text{ not free in tm and } x \text{ not free in } \text{N} \]
\[ = (\forall z \cdot [\text{tm}/([z/x]P)]\text{) if } x \text{ is not free in tm and } x \text{ is free in } \text{N} \text{ and tm is free in } P \text{ and } z \text{ is an introduced variable which is not free in either } \text{N} \text{ or in } P. \]

13.1.5 Simultaneous substitution

We also use \([t_1/x_1, \ldots, t_n/x_n]M\) to mean the term derived from \(M\) by \textit{simultaneous} substitution of \(t_1\) for each free occurrence of \(x_1\), ..., \(t_n\) for each free occurrence of \(x_n\). It is easy to see that, in general, successive substitution is \textit{not} equivalent to simultaneous substitution. As an example, consider the term \(Q(x,y)\).

\[ [x/y][y/x]Q(x,y) = [x/y]Q(y,y) = Q(x,x), \text{ whereas} \]

\[ [y/x,x/y]Q(x,y) = Q(y,x). \]

13.1.6 subst

The most primitive substitution mechanism in HOL is \texttt{subst} which takes a list of pairs \([(t_1, x_1); \ldots; (t_n, x_n)]\) and a term \(M\) and carries out the simultaneous substitution \(M[t_1/x_1, \ldots, t_n/x_n]\). Here are some simple examples which illustrate the effects of \texttt{subst}:

On constants.

```
#subst [ ("0", "1") ] "2 = 1 + 1";;
"2 = 0 + 0" : term
```

On variables.

```
#subst [ ("a:bool", "x:bool") ] "b:bool";;
"b" : term

#subst [ ("a:bool", "x:bool") ] "x:mm";;
"x" : term

#subst [ ("a:bool", "x:bool") ] "x:bool";;
"a" : term
```

On applications.
The next examples give more detail on how subst works. The substitutions are given as a list of (new, old) pairs. subst makes a single pass over the target term carrying out a recursive traversal from left to right and from top to bottom. This means that the substitution that is taken is the "largest", e.g.

At each step subst will compare the current subterm in the target with each "old" term of the pairs in the substitution list in turn (also considered from left to right). If the current subterm is identical to the current "old" term from the substitution list, it is replaced by the corresponding "new" term.

If current subterm cannot be replaced, subst is called recursively over its subterms. As soon as the current subterm can be replaced, the substitution is carried out and that recursive call bottoms out.

On abstractions.
Our final example illustrates the name capture problem. Substituting \( y \) for \( x \) in the lambda term \( \lambda y. x \) must not result in \( \lambda y. y \). Accordingly, an \( \alpha \) conversion is applied to the bound variable and its occurrences.

\[
\text{\#subst [ ("y:bool", "x:bool") ] } "(\lambda y. x)"
\]

\[
"\lambda y'. y y" : \text{term}
\]

### 13.2 Representing HOL theorems

Theorems are represented as sequents—a list of hypotheses (a list of terms) together with a conclusion (a term) that has been inferred.

```plaintext
abstract type thm = term list # term
with
    .......;
```

**Creating theorems.** Theorems are to be proved, not faked. **Taking theorems apart.** Theorems may be taken apart using `dest_thm` which returns a sequent. `hyp th` and `concl th` return only the hypotheses or the conclusion (respectively) of a theorem. Obviously, `hyp = fst o dest_thm` and `concl = snd o dest_thm`.

```plaintext
#dest_thm ;;
- : (thm -> goal)

#(hyp, concl) ;;
((->, ->) : ((thm -> term list) # (thm -> term))
```

### 13.2.1 Substitution in theorems

The most primitive substitution mechanism for HOL theorems is the inference rule `SUBST` which allows us to substitute for selected occurrences of a term in the conclusion of a theorem. Its effect is defined below:

```
SUBST: (thm # term) list -> term -> thm -> thm

[ (A1 |- ti = ui, "vi") ; ... ; (An |- tn = un, "vn") ]
"template"

A |- th(ti, ..., tn)
------------------------
A + A1 + ... + An |- th(ui, ..., un)
```
We explain how it works through two simple examples. Suppose we have two theorems, \( th1 \) and \( th2 \) where

(i) \( th1 \) is an equality of the form \( lhs = rhs \),
(ii) \( th2 \) contains one or more free occurrences of the term \( lhs \) and
(iii) we wish to replace selected occurrences of \( lhs \) in the conclusion of \( th2 \) by \( rhs \). The template is used to locate the occurrences.

Specifically we have the theorem \( th1 = A \vdash x = 7 \) and we wish to replace the first and fourth occurrences of \( x \) in \( th2 \) below by 7.

```
#let th1 = ASSUME "x = 7";;
th1 = (\- x = 7)

#let th2 = REFL "x + (y + z) + x";;
th2 = (\- x + ((y + z) + x) = x + ((y + z) + x))
```

Here is how it is done:

```
#SUBST[th1, "M:num"] "M + ((y + z) + x) = x + ((y + z) + M)" th2;;
  \- 7 + ((y + z) + x) = x + ((y + z) + 7)
#hyp it;;
["x = 7"] : term list
```

Notice that the resulting theorem has the assumption \( x = 7 \), arising from \( th1 \). We have to tell \( SUBST \) precisely which free occurrences are to be substituted. The third argument to \( SUBST \) is the target theorem in which the substitutions will take place. The second argument is a template which copies the conclusion of the third argument replacing those occurrences of the term we are to replace by a marker variable (here \( M \)). The first argument is a list of pairs (we can carry out several simultaneous substitutions). Each pair gives a substitution (in the form of a theorem) and a unique marker (here \( M \)) to be found in the template. \( SUBST \) works by making sure that the template is derivable from the target theorem, and then uses \( subst \) to effect the substitutions in the template.

Here is a more advanced example which shows simultaneous substitutions, the second of which is a term not a variable.

```
#let th3 = ASSUME "(y + z) = x + 1";;
  th3 = (\- y + z = x + 1)

#let th4 = SUBST [(th1, "M:num"), (th3, "M:num")]
   "M + ((x + 1) + x) = x + ((x + 1) + M)"
  th2;;

th4 = (\- 7 + ((x + 1) + x) = x + ((x + 1) + 7))
```
Finally here is an example to show that \texttt{SUBST} is reversible and thus forms the basis for generating tactics.

```plaintext
#hyp th4;;
[*x = 7*]; "y + z = x + 1" : term list
```

Since \texttt{SUBST} is clumsy to use, there are other, more friendly substitution mechanisms derived from it. \texttt{SUBS\_OCCS}. In \texttt{SUBS\_OCCS} we simply number off the occurrences of the term we wish to replace instead of giving a matching template.

```plaintext
#SUBST [(SYM th1, "num"); (SYM th3, "num")] "M + (N + x) = x + (M + N)"
th4;;
.. 1 - x + ((y + z) + x) = x + ((y + z) + x)
```

\texttt{SUBS}. Most of the time we will want to replace all the occurrences of a subterm by its equivalent, in which case there is no need for a template at all and we use \texttt{SUBS}.

```plaintext
#SUBS;;
- : (thm list -> thm) -> thm
#SUBS [ th1 ] th2;;
.. 1 - 7 + ((y + z) + 7) = 7 + ((y + z) + 7)
#SUBS [ th1; th3 ] th2;;
.. 1 - 7 + ((x + 1) + 7) = 7 + ((x + 1) + 7)
```

13.3 Conversions

A \textit{conversion} maps a term to a theorem expressing the equality of that term to another. We have already seen examples using \texttt{num\_CONV}.

```plaintext
#num\_CONV "4";;
1 - 4 = Suc 3
```
13.3. **CONVERSIONS**

Theorems of this sort are used in a variety of contexts to justify the replacement of a particular term by its semantic equivalent. The ML type of a conversion is `term -> thm` or `conv` for short.

Any conversion can be turned into a inference rule or into a tactic. `CONV_RULE` applied to a conversion `conv` produces an inference rule which can replace the conclusion of a theorem with the equivalent term generated by the conversion. `CONV_TAC` applied to a conversion `conv` produces a tactic which can replace a current goal with the equivalent term generated by the conversion. Writing the conversion first makes sure that the dependent rule and tactic work in a consistent and predictable manner. The technique is used to advantage in rewriting—both rewriting rules and rewriting tactics are based upon the same rewriting conversion. It follows that to understand rewriting well, you first need to understand conversions.

Here are examples of some of the more common conversions:

```
#ALPHA_CONV "x:bool "\ x . x \ y";;
1- (\x . x \ y) = (\x . x \ y)

#AND_FORALL_CONV "(! (t:num) . A t \ (t:num) . B t)";;
1- (!t.A t \ (t.B t) = (!t.A t \ B t)

#FORALL_AND_CONV "! (t:num) . A t \ B t";;
1- (!t.A t \ B t) = (!t.A t) \ (t.B t)

#FUN_EQ_CONV "PRED o SUC = I";;
1- (PRED o SUC = I) = (tn.(PRED o SUC)n = I n)

#REWR_CONV ADD_SYM "x + 5";;
1- x + 5 = 5 + x

#SYM_CONV "x = T";;
1- (x = T) = (T = x)

#BETA_CONV "((\ x . x) T) = T";;
evaluation failed BETA_CONV
```

A conversion only applies to the top level of its argument, so that

```
#BETA_CONV "((\ x . x) T) = T";;
evaluation failed BETA_CONV
```

fails because the top level of the term is an equality not an application. In order to apply a conversion further inside a term, we hand the conversion to a **conversional** (a pun on **tactical**) which can apply the conversion to chosen subterms of the original term.
13.3.1 Converting subterms of terms

**SUB_CONV, RATOR_CONV, RAND_CONV, ABS_CONV**

The basic building block in the development of subterm conversions is **SUB_CONV** which applies a conversion to the top level subterms of a HOL term. It is defined by induction:

**SUB_CONV conv tm** where tm is a primitive term.

- tm is a constant c.
  
  **SUB_CONV conv c** ignores conv and returns \( |- c = c \).

```ocaml
#SUB_CONV num_CONV "5" ;;
|- 5 = 5
```

- tm is a variable v.
  
  **SUB_CONV conv v** ignores conv and returns \( |- v = v \).

```ocaml
#SUB_CONV num_CONV "x:num" ;;
|- x = x
```

- tm is an application f g.
  
  if conv f returns \( |- f = f' \) and conv g returns \( |- g = g' \) then **SUB_CONV conv (f g)** returns \( |- (f g) = (f' g') \).

```ocaml
#SUB_CONV BETA_CONV "((\x . x) SUC)((\x . x)2)" ;;
|- (\x . x)SUC((\x . x)2) = SUC 2
```

- tm is an abstraction \( \ \ x . \ \ body \).
  
  if conv b returns \( |- b = b' \), then **SUB_CONV conv (\ x . b)** returns \( |- (\ x . b) = (\ x . b') \)

```ocaml
#SUB_CONV BETA_CONV "\ x . (\ y . y + x) 2" ;;
|- (\x . (\ y . y + x)2) = (\x . 2 + x)
```

- Otherwise, **SUB_CONV conv tm** fails.

```ocaml
#SUB_CONV num_CONV "\ x . (\ y . y + x) 2";;
evaluation failed    num_CONV: argument not a numeral
```
Other useful conversion operators, all of which are of type $\text{conv} \rightarrow \text{conv}$, are $\text{RATOR}_{\text{CONV}}$, $\text{RAND}_{\text{CONV}}$, and $\text{ABS}_{\text{CONV}}$.

$\text{RATOR}_{\text{CONV}} \text{ conv } \text{tm}$ converts the operator of an application using $\text{conv}$; $\text{RAND}_{\text{CONV}} \text{ conv } \text{tm}$ converts the operand of an application using $\text{conv}$; $\text{ABS}_{\text{CONV}} \text{ conv } \text{tm}$ converts the body of an abstraction using $\text{conv}$.

Because binary expressions are held as combinations, we may selectively penetrate and convert any one of their constituents using $\text{rator}$ and $\text{rand}$.

13.3.2 Combining forms for conversions

Conversions may be applied repeatedly, in sequence or in case of failure by combining them with $\text{REPEATC}$, $\text{THENC}$, and $\text{ORELSEC}$. For example
Two other interesting conversions are **ALL_CONV** and **NO_CONV**. For any term \( \text{tm} \), **ALL_CONV \, \text{tm}** returns \( |- \, \text{tm} = \text{tm} \) and **NO_TAC \, \text{tm}** fails.

The ML code for the three combining forms is straightforward and instructive.

(c1 \text{THENC} c2) \, \text{tm} works in three steps. It first evaluates c1 \, \text{tm} which returns a theorem, say \( \text{th1} = |- \, \text{tm} = T1 \). It then evaluates c2 \, T1 which returns the theorem, say \( \text{th2} = |- \, \text{T1} = T2 \). Finally, from the transitivity of equality it infers that \( |- \, \text{tm} = T2 \). (c1 \text{THENC} c2) \, \text{tm} fails if either of the first two steps fails.

(c1 \text{ORELSEC} c2) \, \text{tm} works in two steps. It first evaluates c1 \, \text{tm}. If step 1 returns a theorem, say \( \text{th1} = |- \, \text{tm} = T1 \), then (c1 \text{ORELSEC} c2) \, \text{tm} returns \( \text{th1} = |- \, \text{tm} = T1 \). If step 1 fails then (c1 \text{ORELSEC} c2) \, \text{tm} returns c2 \, \text{tm}.

**REPEATC \, \text{conv} \, \text{tm}** applies **THENC \, \text{conv}** repeatedly until it fails and then exits gracefully (taking the last useful result with it) via **ALL_CONV**.
13.3. CONVERSIONS

13.3.3 Depth conversions

Now we step beyond the analogy with tactics and examine conversions that traverse terms recursively and apply a conversion to every subterm. The basic function is \texttt{DEPTH\_CONV}. \texttt{DEPTH\_CONV conv tm} applies the conversion \texttt{conv} to each and every subterm in the representation of \texttt{tm} and replaces each subterm \texttt{t1} by \texttt{t2} if \texttt{t1} converts to \texttt{t2} under \texttt{conv}. The subterm \texttt{t1} is left unchanged if \texttt{conv t1} fails. The search strategy is implemented bottom up and from left to right, visiting each node but once. Here are some examples using \texttt{DEPTH\_CONV}.

\begin{verbatim}
DEPTH\_CONV;;
- : (conv \rightarrow conv)

DEPTH\_CONV BETA\_CONV "(\ x . (\ z . z = 1) x) = T";;
1- ((\x . (\z . z = 1)x) = T) = ((\x . x = 1) = T)

DEPTH\_CONV BETA\_CONV "(\ f . (\ x . f x)) (\ z . z = 1)";;
1- ((\f x . f x)(\z . z = 1) = (\x . (\z . z = 1)x)

DEPTH\_CONV num\_CONV "(\ x . (\ y . x = y) 2) 3";;
1- ((\x . (\y . x = y)2)3 = (\x . (\y . x = y)(SUC 1))(SUC 2)

REDEPTH\_CONV num\_CONV "(\ x . (\ y . x = y) 2) 3";;
1- ((\x . (\y . x = y)2)3 = (\x . (\y . x = y)(SUC(SUC 0)))(SUC(SUC(SUC 0))))
\end{verbatim}

Because \texttt{DEPTH\_CONV} does not reapply itself to the subterms it replaces, it cannot cope with nested conversions. \texttt{REDEPTH\_CONV conv tm} is more sophisticated and calls itself on any new subterm it generates.

\begin{verbatim}
REDEPTH\_CONV;;
- : (conv \rightarrow conv)

REDEPTH\_CONV BETA\_CONV "(\ f . (\ x . f x)) (\ z . z = 1)";;
1- ((\f x . f x)(\z . z = 1) = (\x . (\z . z = 1)x)

REDEPTH\_CONV BETA\_CONV "(\ f . (\ x . f x)) (\ z . z = 1)";;
1- ((\f x . f x)(\z . z = 1) = (\x . (\z . z = 1)x)

REDEPTH\_CONV num\_CONV "(\ x . (\ y . x = y) 2) 3";;
1- ((\x . (\y . x = y)2)3 = (\x . (\y . x = y)(SUC 1))(SUC 2)

REDEPTH\_CONV num\_CONV "(\ x . (\ y . x = y) 2) 3";;
1- ((\x . (\y . x = y)2)3 = (\x . (\y . x = y)(SUC(SUC 0)))(SUC(SUC(SUC 0))))
\end{verbatim}

Here is a possible ML definition for \texttt{REDEPTH\_CONV}:

\begin{verbatim}
letrec REDEPTH\_CONV conv tm
  = \tm : (sub\_conv
     (REDEPTH\_CONV conv)
   THENC \conv \THENC \REDEPTH\_CONV conv)
   \tm ;;
REDEPTH\_CONV = \tm : (conv \rightarrow conv)
\end{verbatim}
DEPH\_CONV and REDEPHT\_CONV work from the bottom up and apply to subterms before applying to top-level terms. TOP\_DEPTH\_CONV works from the top down, and applies the conversion to a term before its subterms.

```plaintext
#letrec TOP\_DEPTH\_CONV conv tm
  = (REPEATC conv
     THEMEC
       (SUB\_CONV
         (TOP\_DEPTH\_CONV conv)
         THEMEC (conv THEMEC (TOP\_DEPTH\_CONV conv)))
     ORELSEC ALL\_CONV
     ) tm;

TOP\_DEPTH\_CONV = - : (conv -> conv)
```

REDEPTH\_CONV and TOP\_DEPTH\_CONV do more searching and more reduction than DEPTH\_CONV because they replace every subterm \t_1\ by \t_2' if \t_1\ reduces to \t_2\ by conv and \t_2\ recursively reduces to \t_2'\ by TOP\_RE\_DEPTH\_CONV conv.

Divergence in conversions

DEPH\_CONV, REDEPHT\_CONV and TOP\_DEPTH\_CONV may all diverge, if the conversion may be successfully reapplied to each new term. The operator CHANGED\_CONV makes a conversion fail if applying it leaves the term unchanged. That is if conv tm evaluates to \!\ |- \!\ tm = tm'\ then CHANGED\_CONV conv tm also evaluates to \!\ |- \!\ tm = tm'\ unless tm is \!-equivalent to tm' in which case it fails. We always include it in calls on DEPTH\_CONV, REDEPHT\_CONV and TOP\_DEPTH\_CONV just to be on the safe side, writing for example

```plaintext
DEPH\_CONV (CHANGED\_CONV (FORALL\_AND\_CONV))
```

instead of

```plaintext
DEPH\_CONV (FORALL\_AND\_CONV)
```

Finally ONCE\_DEPH\_CONV operates like DEPH\_CONV except that it exits after its first successful conversion in a subterm. ONCE\_DEPH\_CONV cannot diverge. Use it if it does the job since it is faster.

```plaintext
#ONCE\_DEPH\_CONV BETA\_CONV "((\ f . f + 1) 2) + (\ g . (\ f . f+1)2)3"; ;
\!\!\ |- ((\ f . f + 1)2) + (\ g . (\ f . f+1)2)3) = (2 + 1) + ((\ f . f + 1)2)

#DEPH\_CONV BETA\_CONV "((\ f . f + 1) 2) + (\ g . (\ f . f+1)2)3"; ;
\!\!\ |- ((\ f . f + 1)2) + (\ g . (\ f . f+1)2)3) = (2 + 1) + (2 + 1)
```
Because these conversion forms are used often to create tactics and rules from conversions, we define a useful set of functions to provide selected traversals of the conversion. \texttt{CHANGED\_CONV} is not needed in the last two since \texttt{ONCE\_DEPTH\_CONV} cannot diverge.

\begin{verbatim}
#let DEPTH\_CONV\_TAC = CONV\_TAC o DEPTH\_CONV o CHANGED\_CONV
and DEPTH\_CONV\_RULE = CONV\_RULE o DEPTH\_CONV o CHANGED\_CONV
and REDEPTH\_CONV\_TAC = CONV\_TAC o REDEPTH\_CONV o CHANGED\_CONV
and REDEPTH\_CONV\_RULE = CONV\_RULE o REDEPTH\_CONV o CHANGED\_CONV
and TOP\_DEPTH\_CONV\_TAC = CONV\_TAC o TOP\_DEPTH\_CONV o CHANGED\_CONV
and TOP\_DEPTH\_CONV\_RULE = CONV\_RULE o TOP\_DEPTH\_CONV o CHANGED\_CONV
and ONCE\_DEPTH\_CONV\_TAC = CONV\_TAC o ONCE\_DEPTH\_CONV
and ONCE\_DEPTH\_CONV\_RULE = CONV\_RULE o ONCE\_DEPTH\_CONV;;

DEPTH\_CONV\_TAC = - : (conv ➔ tactic)
DEPTH\_CONV\_RULE = - : (conv ➔ thm ➔ thm)
REDEPTH\_CONV\_TAC = - : (conv ➔ tactic)
REDEPTH\_CONV\_RULE = - : (conv ➔ thm ➔ thm)
TOP\_DEPTH\_CONV\_TAC = - : (conv ➔ tactic)
TOP\_DEPTH\_CONV\_RULE = - : (conv ➔ thm ➔ thm)
ONCE\_DEPTH\_CONV\_TAC = - : (conv ➔ tactic)
ONCE\_DEPTH\_CONV\_RULE = - : (conv ➔ thm ➔ thm)
\end{verbatim}

13.3.4 Tools for hardware verification

There are some special tools supplied to help us with hardware verification, specifically with removing hidden lines. They are stored in the library \texttt{unwind} which needs to be loaded before they can be accessed.

\begin{verbatim}
#load_library `unwind';;
Loading library unwind ...
Updating help search path
............................................
Library unwind loaded.
() : void
\end{verbatim}

There is one dot for each definition in the library. We will only describe two of these definitions—\texttt{UNWIND} and \texttt{PRUNE}. When carrying out hardware verifications, we typically try to prove that \texttt{imp = spec} over all inputs and outputs. We start by setting up a goal and by rewriting with the implementation and the specification definitions. To be concrete we take the example of a line of 4 inverters connected together through hidden wires \(p, q\) and \(r\) and with input \(i\) and output \(z\). It is easy to unfold the implementation to the term \texttt{imp'} below

\begin{verbatim}
#let imp' =
  " ? p q r . (p = ~i) /
  (q = ~p) /
  (r = ~q) /
  (z = ~r)
";;
imp' = " ? p q r . (p = ~i) /
  (q = ~p) /
  (r = ~q) /
  (z = ~r) : term
\end{verbatim}
In general there is one defining equation for each hidden line and for each device output line. Each defining equation has the form \( v = \text{rhs} \). \text{rhs} is the value on the line \( v \) as determined by the implementation definition. If the line represented by \( v \) is hidden then \( v \) will be existentially quantified. In combinational circuits, the right hand sides will be terms involving input lines and hidden lines only and no line may appear in its own right hand side\(^2\). The built-in conversion \text{UNWIND} takes such a term as \text{imp}\(^1\) and for each hidden line \( v \) in turn uses its defining equation \( v = \text{rhs} \) to substitute for each occurrence of the name in each and every \text{rhs} throughout the term. After one application of \text{UNWIND} hidden line names have been eliminated from all the right hand sides which now contain only input line names.

\[
\text{UNWIND.AUTO_CONV} \text{imp'};;
\]
\[
\vdash (p \ q \ r. \ (p = \cdash i) \ \land \ (q = \cdash p) \ \land \ (r = \cdash q) \ \land \ (z = \cdash r)) =
(p \ q \ r. \ (p = \cdash i) \ \land \ (q = \cdash i) \ \land \ (r = \cdash i) \ \land \ (z = \cdash i))
\]

The conversion \text{PRUNE_CONV} takes a term of this form and removes the now trivially satisfiable equations for the hidden lines. Thus if we first apply \text{UNWIND.AUTO_CONV} and then \text{PRUNE_CONV} to a term like \text{imp}\(^1\) we can substitute for and then remove hidden lines.

\[
\text{PRUNE_CONV} \text{imp'};;
\]
\[
\vdash (p \ q \ r. \ (p = \cdash i) \ \land \ (q = \cdash p) \ \land \ (r = \cdash q) \ \land \ (z = \cdash r)) = (z = \cdash i)
\]

Here is a general conversion based upon what we have done.

\[
\text{DEPTH_CONV} \text{imp'};;
\]
\[
\vdash (p \ q \ r. \ (p = \cdash i) \ \land \ (q = \cdash p) \ \land \ (r = \cdash q) \ \land \ (z = \cdash r)) = (z = \cdash i)
\]

It is now a trivial matter to write \text{EXISTS_ELIM_RULE} and \text{EXISTS_ELIM_TAC}.

\[
\text{EXISTS_ELIM_RULE} = \text{DEPTH_CONV_RULE} \text{imp'};;
\]
\[
\text{EXISTS_ELIM_TAC} = \text{DEPTH_CONV_TAC} \text{imp'};;
\]

Example 13.3.3 \text{IMP_ANTISYM_CONV}. Write a conversion to take the term "\( a \implies b \ \land \ b \implies a \)" to the theorem "\( \vdash (a \implies b \ \land \ b \implies a) = (a = b) \)".

We recall the two forward inference rules

\(^2\)Feedback may occur in sequential circuits but then signals are distinguished by different time stamps.
and build towards the conversion by assuming a term of the expected format

```plaintext
#let tm = "((a ==> b) \ (b ==> a))";;
```

Since we have an equality \( \vdash \text{lhs} = \text{rhs} \) to prove, we first show that \( \vdash \text{lhs} ==> \text{rhs} \). Then we show that \( \vdash \text{rhs} ==> \text{lhs} \). Finally we apply **IMP_ANTISYM_RULE**.

To show that \( ((a ==> b) \land (b ==> a)) ==> (a = b) \) we assume \( \text{tm} \), take its conclusion apart, and then use **IMP_ANTISYM_RULE** to construct a new theorem with conclusion \( a = b \).

```plaintext
#let th = ASSUME tm;;
```

The second part is no harder. In order to show that \( (a = b) ==> ((a ==> b) \land (b ==> a)) \) we pick up "\( a = b \)" as the conclusion of \( \text{lhs1} \), take it apart with **EQ_IMP_RULE** and then conjoin these parts.

```plaintext
#let (l, r) = EQ_IMP_RULE (ASSUME(concl lhs1));
```

```plaintext
#let rhs1 = (a ==> b) \ (b ==> a);;
```

```plaintext
#let rhs2 = DISCH_ALL rhs1;;
```

```plaintext
\( \vdash (a ==> b) \land (b ==> a) = (a = b) \)
```
All that is left is to gather these parts together as an ML function.

```ml
#let IMP_ANTI_SYM_CONV tm
  = (let th = ASSUME tm
    in
    let lhs = IMP_ANTI_SYM_RULE (CONJUNCT1 th) (CONJUNCT2 th) in
    let (l, r) = EQ_IMP_RULE (ASSUME(concl lhs)) in
    let rhs = CONJ l r in
    IMP_ANTI_SYM_RULE (DISCH_ALL lhs) (DISCH_ALL rhs)
  ) ? failwith 'EQ_IMP_CONV';;

IMP_ANTI_SYM_CONV = - : conv
```

and then complete our work by making a rule and a tactic from the conversion.

```ml
#let IMP_ANTI_SYM_RULE
  = (DEPTH_CONV_RULE IMP_ANTI_SYM_CONV) ? failwith 'IMP_ANTI_SYM_RULE';;
IMP_ANTI_SYM_RULE = - : (thm -> thm)

#let IMP_ANTI_SYM_TAC
  = (DEPTH_CONV_TAC IMP_ANTI_SYM_CONV) ? failwith 'IMP_ANTI_SYM_TAC';;
IMP_ANTI_SYM_TAC = - : tactic
```

Here are two examples of `IMP_ANTI_SYM_TAC` in action

```ml
#g "(a \ b => c) \ (c => a \ b)";;
"(a \ b => c) \ (c => a \ b)"

() : void
```

```ml
#e (IMP_ANTI_SYM_TAC);;
OK.
"a \ b = c"

() : void
```

```ml
#g "(a \ b => (((p \ q) => r) \ (r => p \ q))) \ a \ b)";;
"(a \ b => (((p \ q) => r) \ (r => p \ q))) \ a \ b)"

() : void
```

```ml
#e (IMP_ANTI_SYM_TAC);;
OK.
"a \ b = (p \ q = r)"

() : void
```
13.4 Forward inference

Forward inferencing is a viable technique when a proof is small and needs no bookkeeping (a situation that arises often at the leaves of proof trees). As an example, here is the forward proof of \(((2 \text{ EXP } (\text{SUC } n)) = \text{val} a \ n\) in which we use the inference rules MATCH, MP and GSYM.

```
MATCH_MP LESS_NOT_EQ (SPEC_ALL maxword);
|- \!n a. (val a n < (2 EXP (SUC n)))
: (thm # thm)

MATCH_MP LESS_NOT_EQ (SPEC_ALL maxword);
|- \! (val a n = 2 EXP (SUC n))

MATCH_MP LESS_NOT_EQ (SPEC_ALL maxword);
|- \2 EXP (SUC n) = val a n
```

13.4.1 Primitive inference rules

The 8 primitive inference rules of HOL are listed in appendix B. They are

- ASSUME, BETA_CONV, DISCH, MP, REFL, SUBST which we have met before, and
- ABS, INST_TYPE which are new.

Here are the definitions and sample applications of ABS and INST_TYPE.

**ABS**

```
ABS : term -> thm -> thm
"x"

A |- t1 = t2

-------------
A |- (\ x . t1) = (\ x . t2)
```

```
ABS "x:*" (REFL "x:*" )
|- (\ x . x) = (\ x . x)
```

**INST_TYPE.** One of the strengths of HOL is the generality ensuing from its use of polymorphism. We can prove theorems that are valid over all types. However when we come to use the theorem in a specific case we have to type-instantiate it first since substitution only works for identical types.
For example, \( \text{EQ\_SYM\_EQ} \) is defined for \( x \) and \( y \) being of type 
":*". The attempt below to specialise this theorem fails because we can only substitute \underline{like} type for \underline{like} type.

```plaintext
#SPECL ["3";"4"] EQ\_SYM\_EQ;
```

We must first produce a \textit{num} form of \texttt{EQ\_SYM\_EQ} and then the specialisation goes through.

```plaintext
#SPECL ["3";"4"] (INST\_TYPE [":num",":*"] EQ\_SYM\_EQ);
```

\textbf{Example 13.4.4 Proof of \( \lnot F \)}

We now present a forward proof (due to Tom Melham) which uses five of the primitive inference rules. The negation operator is defined in HOL by:

\[ \text{NOT\_DEF} = \lnot \$ = (\forall t. t => F) \]

The \underline{top-down} (i.e. backward proof) chain of thought behind the proof is roughly: given the goal \( \lnot F \), replace the operator \( \lnot \) by its definition. This yields a new goal \( \forall t. t => F \) which is \( \beta \)-equivalent to \( F => F \). But this goal is easy to prove. Here is Tom’s forward proof:
Perhaps you can see why we chose to work with $\lambda$HOL in part I!

### 13.4. Derived inference rules

Other common forward inferencing rules are:

- GEN, GEN\_ALL, SPEC, SPEC\_ALL
- CHOOSE, EXISTS
- EQ\_IMP\_RULE, IMP\_ANTISYM\_RULE
- CONJUNCT1, CONJUNCT2, CONJ
- DISJ1, DISJ2, DISJ\_CASES
- DISCH, UNDISCH

which we have met before as inference rules in $\lambda$HOL,

- SYM, SYM\_RULE
which we used in part III, and

- **DISCH\_ALL, UNDISCH\_ALL**
- **CONJUNCTS, CONJ\_LIST**

which are new. Here are their descriptions:

- **UNDISCH\_ALL** `thm` takes a theorem in the form of an implication, say `a ==> b`, and places the antecedent `a` on the assumption list. The process is repeated if `b` is an implication. E.g.,
  \[
  \text{UNDISCH\_ALL } \vdash a \Rightarrow (b \Rightarrow (c \Rightarrow d)) = ["a";"b";"c"] \vdash d.
  \]

- **DISCH\_ALL** `thm` takes a theorem and discharges all its assumptions. The order is unexpected. E.g.,
  \[
  \text{DISCH\_ALL } (["a";"b";"c"] \vdash d) = \vdash (c \Rightarrow (b \Rightarrow (a \Rightarrow d))).
  \]
  Thus, even if a theorem has no assumptions initially,
  \[
  \text{DISCH\_ALL o UNDISCH\_ALL } \neq I.
  \]
  Pity.

- **CONJUNCTS** `thm` takes a theorem whose top level form is a number of conjuncts and returns a list of the conjuncts as individual theorems.

  \[
  \text{CONJUNCTS: thm } \rightarrow \text{ thm list}
  \]

  \[
  A \vdash t_1 \land (t_2 \land (\ldots \land t_m)\ldots)
  
  \]

  \[
  \hspace{2cm}
  \frac{}{[A \vdash t_1; A \vdash t_2; \ldots; A \vdash t_m]}
  \]

  A nice application of **CONJUNCTS** is to use it and the function `el` to select a specific rewriting agent from a conjunction of theorems. For example, should we wish to rewrite with the third of the four theorems of **ADD\_CLAUSES** we can select it by

  \[
  \text{#el 3 (CONJUNCTS ADD\_CLAUSES);}.
  \]

  \[
  \vdash (\text{SUC } m) + n = \text{SUC}(m + n)
  \]

- **LIST\_CONJ** `[ thm ]` takes a list of theorems and returns one theorem that conjoins them.

  \[
  \text{LIST\_CONJ: proof}
  \]

  \[
  \frac{}{[A_1 \vdash t_1; A_2 \vdash t_2; \ldots; A_n \vdash t_n]}
  
  \]

  \[
  \hspace{2cm}
  \frac{}{[A_1+A_2+\ldots+A_n \vdash (t_1 \land t_2 \land \ldots \land t_n)]}
  \]
A typical use of \texttt{CONJUNCTS} and \texttt{LIST\_CONJ} occurs when we have a developed a theorem with several conjoined cases each of which we wish to transform. As a concrete example, suppose we wish to apply the symmetric and generalised versions of the theorems in \texttt{ADD\_CLAUSES}. We put them in a list via \texttt{CONJUNCTS}, and map the required rules down the list. We could, if we wanted, rewrite with this list.

\begin{verbatim}
#map (GEN\_ALL o SYM) (CONJUNCTS ADD\_CLAUSES);;
|   |- !m. m = 0 + m;
|   |- !m. m = m + 0;
|   |- !m n. SUC(m + n) = (SUC m) + n;
|   |- !m n. SUC(m + n) = m + (SUC n)]
: thm list
\end{verbatim}

or we could transform it back to a single theorem with \texttt{LIST\_CONJ}.

\begin{verbatim}
#LIST\_CONJ it;;
|   |- (!m. m = 0 + m) /
|   |- (!m. m = m + 0) /
|   |- (!m n. SUC(m + n) = (SUC m) + n) /
|   |- (!m n. SUC(m + n) = m + (SUC n))
\end{verbatim}

### 13.5 Rewriting

Rewriting can be performed on both theorems and goals, and there are equivalent suites of rewriting rules and rewriting tactics. We shall focus initially on rewriting rules, but most of what is said applies equally to the corresponding rewriting tactics.

Rewriting differs from substitution in that it copes with pattern matching and bound variables. Rewriting uses equational theorems, typically of the form $\forall a b \ldots z. \text{lhs} = \text{rhs}$, to replace all terms that can be pattern matched to $\text{lhs}$ by a correspondingly pattern matched $\text{rhs}$.

\texttt{PURE\_REWRITE\_RULE} : (thm list $\rightarrow$ thm $\rightarrow$ thm) is likely the most used of the rewrite rules. The argument is a list of theorems to be used as left-to-right rewriting agents. Typical forms of rewriting agents are:

- $| -. !a b c \ldots z . \text{lhs} = \text{rhs}$
- $| -. \text{lhs1} = \text{rhs1} \land \ldots \land \text{lhsn} = \text{rhn}$
- $| -. \text{tm}$, which is treated as $| -. \text{tm} = T$
- $| -. \sim\text{tm}$, which is treated as $| -. \text{tm} = F$
In every case, all the free variables in the rewriting equations are generalised before any rewriting is done. Then rewriting is carried out on all the subterms of the theorem to be rewritten, and repeatedly until no further rewrite applies. The theorems in the argument list are taken in an order that is implementation defined, and as a result the order of the rewrites may be hard to predict, if the lhs of two different theorems pattern match the same subterm of the theorem being rewritten. If in doubt, rewrites may be performed sequentially with smaller lists of theorems rather than all at once.

The other rewriting rules are variants on this theme.

1. Rules without the PURE prefix, such as \texttt{REWRITE\_RULE}, also use a standard set of simplifying theorems of the calibre of $\langle x \cdot (x = x) = T$ and $\langle t \cdot (T \land t) = t$ as rewriting agents. A full list of the built-in rewrites is tabulated in Appendix A.

2. Rules labelled with \texttt{ONCE}, such as \texttt{PURE\_ONCE\_REWRITE\_RULE}, terminate their exploration of a subterm after the first rewrite occurs. This does not mean that rewriting terminates: it means that the next subterm is examined. Calls on \texttt{PURE\_ONCE\_REWRITE\_RULE} cannot give rise to infinite looping.

3. Although most of the theorems we use for rewriting have empty assumption lists, this is not always the case. Note that the rules labelled with \texttt{ASM} such as \texttt{PURE\_ASM\_REWRITE\_RULE} will rewrite using both the assumptions of the theorem being rewritten and the conclusions of the rewriting agent theorems as witnessed below.

```plaintext
#ADD\_ASSUM " (x < y) \lor (x > y) = \lnot (x = y)"
  (ASSUME " (x < y) \lor (x >= y)"));
.. 1- x < y \lor x >= y

#PURE\_REWRITE\_RULE [GREATER\_OR\_EQ] it;;
.. 1- x < y \lor x > y \lor (x = y)

#PURE\_ASM\_REWRITE\_RULE [DISJ\_ASSOC] it;;
.. 1- \lnot (x = y) \lor (x = y)

why it;;
["x < y \lor x > y = \lnot (x = y)"; "x < y \lor x >= y"] : term list
```

The full complement of high level rewriting rules include the following:

- \texttt{REWRITE\_RULE}
As a consequence of the repetition by all except the ONCE varieties of rewrite rules, rewriting may get into an infinite loop. Classic examples of rewriting agents that diverge are the symmetric rules

```
(#( ADD_SYM, CONJ_SYM, DISJ_SYM ));
( |- !m n. m + n = n + m, 
  |- !t1 t2. t1 \( t2 = t2 \) \( t1, 
  |- !t1 t2. t1 \( t2 = t2 \) \( t1) 
  : (thm thm thm)
```

which, if they can be applied in one direction, must also work in the reverse direction. When looping would occur, revert to ONCE_REWRITE_RULE or use substitution.

Because of its extra power, rewriting is very much slower than substitution, hence PURE and ONCE varieties of rewriting are provided. It pays to use the cheapest one you can get away with. We want this power most of the time, but there are times when rewriting does too much and carries out changes we do not want, or even diverges. For example, consider

```
(#(ADD_SYM, EQ_MONO_ADD_EQ));
( |- !m n. m + n = n + m, |- !m n p. (m + p = n + p) = (m = n))
  : (thm thm)
```

PURE_REWRITE_RULE would loop forever and PURE_ONCE_REWRITE_RULE reverses both summed terms. To reverse but one of the summed terms using substitution the variables must be free. Below we explicitly pattern match ADD_SYM and specialise EQ_MONO_ADD_EQ.
Implementing rewrite rules

How are these rewrite rules implemented in HOL? As you would by now expect, it is all done using conversions (which explains the consistent behaviour of rewrite rules and rewrite tactics). The primitive rewriting agent is \texttt{REWR_CONV}.

A typical call \texttt{REWR_CONV \textit{th \textit{tm}}} takes a theorem and term as arguments. The first argument, say \( \vdash u = v \), is an equational theorem (possibly quantified) and is used as the rewriting agent. The second argument, say \( t \), is the target term. The call returns a theorem \( \vdash t = t' \) where \( t \) is an instance of \( u \) by type and/or variable instantiation, and \( t' \) is the corresponding instance of \( v \). For example,

\begin{verbatim}
#ADD1;;
l- !m. SUC m = m + 1
#REWR_CONV ADD1 "SUC 0";;
l- SUC 0 = 0 + 1
\end{verbatim}

Observe that \texttt{REWR_CONV} operates only on the top level of the term, and uses only a single (possibly quantified) equational theorem as a rewriting agent. All subterms of a term \( t \) can be rewritten according to an equation \textit{th1} using

\begin{verbatim}
DEPTH_CONV (CHANGED_CONV (REWR_CONV th1));;
\end{verbatim}

as shown below

\begin{verbatim}
DEPTH_CONV (CHANGED_CONV (REWR_CONV ADD1)) "SUC(SUC 0) = PRE(SUC 2)";; l- (SUC(SUC 0) = PRE(SUC 2)) = ((0 + 1) + 1 = PRE(2 + 1))
\end{verbatim}

The next step is the definition of the primitive rewriting function named \texttt{GEN_REWRITE_RULE}.

\begin{verbatim}
GEN_REWRITE_RULE;;
- : (conv \textit{\rightarrow} conv) \textit{\rightarrow} \textit{thm list} \textit{\rightarrow} \textit{thm list} \textit{\rightarrow} \textit{thm}
\end{verbatim}
This rule takes a “search strategy” (type conv→conv), two lists of rewriting agent theorems, and a theorem to be rewritten. The search strategy will simply direct how the rule will descend into the conclusion of the theorem applying REWR_CONV, using functions such as TOP_DEPTH_CONV as you have already seen. The rewriting agent theorems are divided into separate lists for the built-in rewrites and the user supplied theorems. All the rewriting theorems are preprocessed to put them into a sort of canonical rewrite agent form, universally quantifying all free variables, and introducing $\equiv T$ for nonequational conclusions, and $\equiv F$ for negated conclusions. The details of how the set of possible matches is generated is beyond what we wish to describe here. However, it is easy to see that the definition of the various flavours of rewrite rules becomes trivial using this basic rule.

\[
\begin{align*}
\text{let} & \quad \text{PURE_REWRITE_RULE} = \text{GEN_REWRITE_RULE TOP_DEPTH_CONV} [] \\
\text{and} & \quad \text{REWRITE_RULE} = \text{GEN_REWRITE_RULE TOP_DEPTH_CONV} \\
\text{and} & \quad \text{PURE_ONCE_REWRITE_RULE} = \text{GEN_REWRITE_RULE ONCE_DEPTH_CONV} [] \\
\text{and} & \quad \text{ONCE_REWRITE_RULE} = \text{GEN_REWRITE_RULE ONCE_DEPTH_CONV} \\
\text{let} & \quad \text{PURE_ASM_REWRITE_RULE th1 th} = \\
& \quad \text{PURE_REWRITE_RULE ((map ASSUME (hyp th)) @ th1) th;} \\
\end{align*}
\]

Thus all the rewriting rules are built from the general purpose rule GEN_REWRITE_RULE, which in turn is defined using REWR_CONV. The PURE-rewriting tools in HOL are defined using the conversional ONCE_DEPTH_CONV (for ONCE_REWRITE_RULE) or TOP_DEPTH_CONV (for REWRITE_RULE). Dropping the prefix PURE_ augments the list of rewriting agents by some standard theorems; adding the prefix ASM_ augments the list of rewriting agents by the assumptions.

You will soon discover that the successful and efficient application of rewriting is central to most proofs you will ever undertake in HOL. Thus, knowledge of the operation of the rewriting rules and tactics can be very helpful.

**Rewriting tactics**

In parallel with the rewriting rules, there are varied flavours of rewriting tactics. Instead of operating on a theorem and its hypotheses, they act upon a goal with a list of assumptions. For each rewrite rule, there is a corresponding rewrite tactic, taking the same theorem list argument.

- **REWRITE_TAC**
The only significant difference in actual practice between rewrite rules and rewrite tactics (and indeed, between rules and tactics in general) is that assumptions of theorem arguments to rewrite rules are added to the theorem produced by the rules, if not already present (modulo alpha equivalence), while assumptions of theorem arguments to rewrite tactics which are found on the assumption list of the current goal (modulo alpha equivalence again) will cause the tactic to fail.

In the example, the tactic failed because the hypothesis of the theorem used as the rewriting agent did not precisely match an assumption of the goal, although both conjuncts appeared separately. A rewrite rule however simply adds hypotheses which don’t already appear.
EXERCISES 13

Exercise 13.1 Define a data type for HOL terms and encode the algorithms for free and bound variables. Then encode the variable substitution algorithm.

```ml
letrec free x tm
  = ( let c = dest_const tm in false)
  ? ( let y = dest_var tm in (x = y))
  ? ( let (l, r) = dest_comb tm in (free l) or (free r))
  ? ( let (y, body) = dest_abs tm
       in
       let qvar = dest_var y
       in
       not (x = qvar) & (free x body)
  )
  false;;
```

Exercise 13.2 Write a function to check that two HOL terms are identical, then repeat the above exercise for term substitution.
Chapter 14

Tactics and tacticals

In this chapter we survey and catalogue all the tactics we have used (and will use during the course of this text) and the common tacticals that can be used to weave them together.

14.1 Tactics

A goal is a sequent—a pair consisting of a list of assumptions (a list of terms with no particular ordering) and a conclusion (term). We interpret a goal as having a conclusion which we wish to prove given the terms in the assumption list. Tactics are used to either break a goal into simpler subgoals or to simplify the current goal by rewriting or substitution or by deriving new assumptions. In this section we list the most common tactics divided into groups: tactics particular to goal structure, tactics for case analysis, ways of manipulation assumptions, tactics for utilizing theorems and assumptions, substitution in a goal, rewriting a goal, and others (the left-overs). We have met many of them already. Any useful categorization will of course suffer from an overlap between categories. We have tried to present a classification which will be useful in considering different possible actions when faced with a goal.

In this and the following chapter we use the convention of explicitly showing goals complete with an assumption list (which may be empty), typically \((as1, tm)\).

14.1.1 Tactics related to the structure of the goal

Often the form of the goal itself can suggest what tactic may best be applied. In general, these tactics apply to the top level syntactic construct only of the goal, not to structures contained within the term. We list tactics for conjunctions, disjunctions, implications, negations, and both universally and existentially quantified goals.

Conjunctions and disjunctions

- CONJ_TAC splits a conjunctive goal \((as1, tm1 \land \ lnot \ tm2)\) into two subgoals \((as1, tm1)\) and \((as1, tm2)\) to be solved separately.
CHAPTER 14. TACTICS AND TACTICALS

- **DISJ1_TAC** and **DISJ2_TAC** can be applied to a disjunctive goal $(as1, tm1 \lor tm2)$. **DISJ1_TAC** returns a subgoal consisting of only the first disjunct $(as1, tm1)$, while **DISJ2_TAC** applied to the same goal returns the subgoal consisting of the second disjunct $(as1, tm2)$.

**Equations, implications and negations**

- **EQ_TAC** takes a goal in the form of an equation $(as1, tm1 = tm2)$ and replaces it by two subgoals: one of the form $(as1, tm1 \Rightarrow tm2)$; the other of the form $(as1, tm2 \Rightarrow tm1)$. It is often followed by a call on **STRIP_TAC THEN ASM_REWRITE_TAC [ thms ]**.

- **REFL_TAC** solves the special case of a symmetric equational goal $(as1, tm = tm)$.

- **DISCH_TAC** moves the antecedent of an implicational goal onto the assumption list. Note that it treats the conclusion $G$ as though it were $G \Rightarrow F$.

```
#g "x <= y => ~((x = y))";
"x <= y => ~((x = y))"
()
: void

#expand (DISCH_TAC);
OK.
" ~((x = y))"
[ "x <= y" ]
()
: void

#expand (DISCH_TAC);
OK..
" ~((x = y))"
[ "x <= y" ]
[ "x = y" ]
()
: void
```

- The inverse **UNDISCH_TAC** takes a term, say tm, on the assumption list, and returns a new conclusion for the goal $(as1, tm \Rightarrow G)$ and deletes tm from the assumptions. **UNDISCH_TAC tm** fails if tm is not an assumption. Often, explicit type information will have to be included in the term. We continue with the previous goal example.
Quantified goals

- **GEN_TAC** expects the top level of the goal to be universally quantified, say \((\text{asl} \, !x \, . \, \text{body})\). It returns the new goal \((\text{asl}, \, \text{body})\).

- **X_GEN_TAC** is like **GEN_TAC** but allows you to select the variable used to specialize the goal. **X_GEN_TAC** "\(v\)" takes a goal of the form \((\text{asl}, \, ! q \, . \, G[q])\) to \((\text{asl}, G[v/q])\), provided that the types of the variables match. The variable "\(v\)" must not appear free in the assumptions nor in \(G\).

- The inverse tactic **SPEC_TAC** may be used to generalize a goal. Applied to a term, variable pair \((\text{tm}, v)\), and a goal \((\text{asl}, G)\), the tactic will return the goal with \(v\) substituted for all free instances of \(\text{tm}\) in \(G\), with the variable "\(v\)" universally quantified as well \((\text{asl}, \, ! v \, . \, G[v/\text{tm}])\). It may be used when we wish to generalize a goal and then try induction or else if we wish to reorder the quantified variables.

- **EXISTS_TAC** \(\text{tm}\) applied to a goal with an existential quantifier \((\text{asl}, \, ?x \, . \, P[x])\) will return the goal with \(\text{tm}\) substituted for \(x\) in the body of the expression, i.e. \((\text{asl}, \, P[\text{tm}/x])\). Variable renaming is done automatically if necessary to avoid free variable capture.

```ml
#expand (UNDISCH_TAC "x <= y");;

OK.. "x <= y ==> P";
[ "x = y" ]

() : void
```

```ml
#g "a <= (a + SUC b)";;
"a <= (a + (SUC b))"

() : void
```

```ml
#e (SPEC_TAC ("SUC b:num", "x:num");;
OK.. "!x. a <= (a + x)"

() : void
```

```ml
#e (SPEC_TAC ("a:num", "y:num");;
OK.. "!y x. y <= (y + x)"

() : void
```
Other structure related tactics

- **STRIP_TAC** breaks a goal apart by using one of **CONJ_TAC, DISCH_TAC** or **GEN_TAC**. If the goal is an implication (as1, A ==> B) it applies **DISCH_TAC** and then uses **STRIP_ASSUME_TAC** (see below) to break A into smaller parts. It may cause several subgoals to be generated.

- **BETA_TAC** will reduce every beta-redex in a goal, not only operating at the top level. Bound variable renaming is performed to avoid free variable capture. The example expresses the expanded definition of the identity combinator "I".

```ml
# g "(\f \x y. f x (g x)) (\x y. x) (\x y. x) x = x";;
"((\f g x. f x (g x)) (\x y. x) (\x y. x) x = x"
() : void

# expand (BETA_TAC);;
OK.
"(\x y. x') x((\x y. x) x) = x"
() : void

# expand (BETA_TAC);;
OK.
"x = x"
() : void
```

### 14.1.2 Tactics for case analysis on the goal

Often a goal can be most readily solved by case analysis. The first tactic simply substitutes boolean constants for the selected term within the goal. Sometimes it is important to remember the case on which we are splitting. The subsequent tactics put that case on the assumption list. This allows us to use the case when we rewrite from the assumptions. We can also use the case to generate more useful theorems on the assumption list via **IMP_RES_TAC** and **RES_TAC**.

- **BOOL_CASES_TAC tm** does case analysis on the boolean term **tm**. It generates two subgoals, one with T substituted for **tm**; the other with F substituted for **tm**.

- **ASM_CASES_TAC tm** does cases analysis on **tm**. It generates two subgoals, one with **tm** as an extra assumption; the other with \(\sim tm\) as an extra assumption.
14.1. Tactics

- **COND_CASES_TAC** searches the goal from left to right for a conditional, say \( p \Rightarrow a \mid b \), and then does cases on its IF-part (here \( p \)). It generates two subgoals, one with \( p \) as an extra assumption; the other with \( \neg p \) as an extra assumption.

- **DISJ_CASES_TAC** (1- u \( \lor \) v) splits a goal into two cases: one with \( u \) as an extra assumption; the other with \( v \) as an extra assumption. For example,

```plaintext
#LESS_CASES;;
1- m n. m < n \( \lor \) n \( \Leftarrow \) m

#e "(n < m \( \lor \) m < n)";;
"(n < m \( \lor \) m < n)"

() : void

#e (DISJ_CASES_TAC (SPEC_ALL LESS_CASES));;;
OK..
2 subgoals
"(n < m \( \lor \) m < n)"
[ "m \( \Leftarrow \) n" ]

"(n < m \( \lor \) m < n)"
[ "m < n" ]

() : void
```

- **INDUCT_TAC** splits a goal of the form (assums, ! n . G[n]) into two cases: the base case (assums, G[0/n]); and the inductive step (assums+G(n), G[SUC n/n]) with G[n] as an added assumption.

- **LIST_INDUCT_TAC** effects structural induction on lists by splitting a goal of the form (assums, ! l . G[l]) where \( l \) is a list type variable into two cases: the base case (assums, G[NIL/l]); and the inductive step (assums+G(l), ! h . G[CONS h l/l]).

Other inductions are possible as well using the more general continuation **INDUCT_THEN** described in chapter 15.

14.1.3 Tactics for manipulating assumptions

We may use terms on the assumption list to help prove the current goal. We need to be able to pick out specific assumptions, add new assumptions, and to derive new assumptions from those that are already there.
Although kept as a list of assumptions, the HOL standard looks upon hypotheses of a theorem, and similarly assumptions of a goal, as an unordered set. We take what the implementation gives us and, of course, this may change in the future. If it does, order dependent proofs may fail. Accordingly we do not introduce or use any of the HOL techniques which rely on the order of the assumptions\(^1\). Thus accessing particular assumptions can seem to be surprisingly awkward, considering especially how often we will wish to do it. We propose a consistent method for doing this which has the advantage of being reasonably robust with respect to possible changes in the HOL system.

One observation is appropriate here concerning the dichotomy of theorems and assumptions in proofs. Many tactics which use assumptions cast the assumption as theorems, simply by ASSUME'ing them. When a proof step utilizes such an assumption/theorem, its validity depends on the presence on the assumption list of every hypotheses of the assumption/theorem. Since the assumption is already present, validity (with respect to this condition) is ensured. The gist of this treatment is that assumptions may often be used precisely the same as theorems by tactics. The simplest example is the tactic \texttt{ASM\_REWRITE\_TAC} which uses the list of assumptions as rewriting agent theorems to rewrite the goal.

Thus this section will concentrate on accessing the assumptions in order to manipulate and use them directly. The use of assumptions extends as far as the use of theorems once these techniques are known.

- \texttt{ASSUME \textit{tm}} is our preferred method of selecting an individual assumption \textit{tm} on the assumption list for immediate use. The argument to \texttt{ASSUME} must be textually identical\(^2\) to an assumption already on the list or else it will fail with the message

\begin{quote}
\texttt{evaluation failed Invalid tactic}
\end{quote}

Although other methods exist for selecting a single assumption, to some extent they all depend on the order in which the assumptions are generated and may fail if the implementation changes. Our technique is admittedly rather clumsy, but is robust and does state explicitly which term we are using. Explicit type information is often needed when supplying the term to the tactic.

\(^1\)In fact there are possible subtle effects of order of assumptions used by tactics such as \texttt{ASM\_REWRITE\_TAC}, but we have avoided tactics which depend directly upon ordering of assumptions.

\(^2\)For some theorem tactics merely being alpha-convertible to an assumption suffices.
• **ASSUM_LIST** is our preferred method of accessing all the assumptions. **ASSUM_LIST tac** passes **ASSUME** over the list of current assumptions and hands the resulting list of theorems to **tac**, which is then applied to the goal.

```latex
\begin{verbatim}
#ASSUM_LIST;;
- : ((thm list -> tactic) -> tactic)

#set_goal([ "x = 1" , "y = 2" ] , "x + y = Suc(Suc(0))");
"x + y = Suc(Suc(0))"
  [ "y = 2" ]
  [ "x = 1" ]

() : void

#o(ASSUM_LIST SUBST_TAC);;
OK...
"1 + 2 = Suc(Suc(0))"
  [ "y = 2" ]
  [ "x = 1" ]

() : void

#b();
"x + y = Suc(Suc(0))"
  [ "y = 2" ]
  [ "x = 1" ]

() : void

#o(ASSUM_LIST (SUBST_TAC
  o (map (CONV_RULE (REDEPTH_CONV num_CONV ))))));;
OK..
"(Suc 0) + (Suc(Suc 0)) = Suc(Suc(Suc 0))"
  [ "y = 2" ]
  [ "x = 1" ]

() : void
\end{verbatim}
```

Since we have the assumption list to hand we can manipulate further before applying the tactic. In the example below, we convert the assumptions into standard **SUC** form before substituting.

Even better, below we solve the entire goal in one step by using rewriting to effect the substitutions, so we can also concatenate (with the .) **ADD_CLAUSES** onto the rewrite list before applying **REWRITE_TAC**.
CHAPTER 14. TACTICS AND TACTICALS

#6();
"x + y = \text{SUC(SUC(SUC 0))}"
[ "y = 2"
[ "x = 1"

() : void

#e(ASSUM_LIST(REREWRITE_TAC
  o (\ L . ADD_CLAUSES . L)
  o (map (CONV_RULE(REDEPTH_CONV num_CONV)))));

OK.

goal proved

| - x + y = \text{SUC(SUC 0)}

Previous subproof:

goal proved

() : void

Here are the steps in the evaluation of the last tactic. The assumption list is \([ x = 1 ; y = 2 ]\). In the table below we denote the goal by \(G\) for short.

\[
\text{ASSUM_LIST(REREWRITE_TAC}
  o (\ \ \ L \ . \ ADD_CLAUSES \ . \ L)
  o (map (CONV_RULE(REDEPTH_CONV num_CONV)))) \ G
\implies \text{REREWRITE_TAC} (\(\text{ADD_CLAUSES} ; \ L\)\)
  (map (CONV_RULE(REDEPTH_CONV num_CONV))
  (map ASSUME \[x=1 ; y=2\])) \ G
\implies \text{REREWRITE_TAC} (\(\text{ADD_CLAUSES} ; \ L\)\)
  (\text{\{1 \ |- x=SUC 0 ; 1 \ |- y=SUC(SUC 0)\}}) \ G
\implies \text{REREWRITE_TAC} [ \text{\{ADD_CLAUSES} ; \ L\)\)
  (\text{\{1 \ |- x=SUC 0 ; 1 \ |- y=SUC(SUC 0)\}}) \ G

By using \text{filter} instead of \text{map} on the argument list we can carry out selective rewriting from the assumptions.

#6();
"x + y = \text{SUC(SUC(SUC 0))}"
[ "y = 2"
[ "x = 1"

() : void
The filtering function selects only equations whose left hand side consists of the variable "y".

- **RULE_ASSUM_TAC** rule cycles through the assumption list and replaces each and every assumption \( \text{asm} \) by (\text{rule asm}), unless rule fails when applied to \( \text{asm} \), in which case \( \text{asm} \) is left intact. In the example below, we rewrite the numbers in each assumption into standard SUC form and also reverse each equation. The composition operator allows us to do this in one pass.
The decision as to whether one should leave the assumptions unchanged and work with `ASSUM_LIST` or transform the assumptions and work with `RULE_ASSUM_TAC` is a matter of style rather than substance. Choose the style you feel most comfortable with.

- `FILTER_ASM_REWRITE_TAC` selects terms from the assumption list for rewriting. It takes a predicate through which it filters all the assumptions and then rewrites the conclusion of the goal with only those that pass through the filter. It can be used to choose the order of rewriting in delicate situations and to prevent too much rewriting, for example the assumptions may lead to cyclic rewriting. Very often the predicate will be one of the built-in query functions prefixed by `is_`

Our example is taken from [44, page 122]. Rewriting using all the assumptions causes looping and we have to pick our way along with great care. We begin by showing the goal in progress, and then give a predicate which will be used to filter assumptions, selecting only those assumptions whose left hand side has one of the given variables.

```
% ... %
"(15(SUC t)->13(SUC t)|14(SUC t)) = (in(SUC t)= out t|out t)"
[ "!t. 11 t = '12 t" ]
[ "!t. 12 t = (in t => 11 t | 12 t)" ]
[ "!t. 13 t = ((t = 0) => F | out(t - 1))" ]
[ "!t. 14 t = T" ]
[ "!t. 15 t = ((t = 0) => F | 14(t - 1))" ]
[ "!t. out t = (15 t => 13 t | 14 t)" ]
()
```

```
#let lines names tm
 = ( let (   , body) = dest forall tm in
    let ( lhs, _ ) = dest eq body in
    let ( rator, _ ) = dest comb lhs in
    let ( x, _ ) = dest var rator in
    mem x names
  ) ? false;;
lines = - : (string list -> term -> bool)
```

We unfold the conclusion of the goal with the equations for lines 11, 13, 14 and 15.
We next unfold the conclusion of the goal with the equations for lines 11, 13, 14 and 15.

and then with the equation for 12. We also rewrite with NOT_SUC which simplifies the condition on the left hand side.

Rewriting with the built-in theorem SUC_SUB1 finishes off the proof.
14.1.4 Tactics to use theorems (and assumptions)

Adding assumptions

We augment the assumption list with `ASSUME_TAC, STRIP_ASSUME_TAC, RES_TAC, IMP_RES_TAC` (and also with `DISCH_TAC, and STRIP_TAC` which were described earlier). If the theorems added are used but once, you may prefer to use a theorem continuation (see chapter 15) which enables you to generate the new theorems, use them and then discard them keeping the assumption list uncluttered with assumptions no longer needed.

- `ASSUME_TAC` (`hyps |- cncl`) adds (merges) `cncl` onto the assumption list. It fails if the hypotheses `hyps` are not a subset of the current assumptions of the goal.

- Rather than adding a theorem conclusion to the assumption list, `MP_TAC` `thm` makes the goal into an implication with the theorem conclusion as its antecedent. The effect of this tactic is similar to using `ASSUME_TAC` followed by `UNDISCH_TAC`.

- `STRIP_ASSUME_TAC` `thm` splits `thm` into a list of theorems and adds them to the assumption list. The theorems are obtained from `thm` by recursively stripping off universal quantifiers, breaking conjunctions into separate terms, doing a case split on disjunctions, and eliminating existential quantifiers by choosing an arbitrary variable.
Using implicative theorems

- **IMPRES_TAC thm** does a weak form of “resolution” (more or less modus ponens and pattern matching) between the antecedents of the implicative theorem *thm* and the assumptions on the assumption list. It begins by transforming *thm* into a “canonical” list of implicative theorems. This transformation is complex, and includes the recursive application of the following steps:

\[
\begin{align*}
H \vdash \neg \text{tm} & \quad \rightarrow \quad H \vdash \text{tm} \implies F \\
H \vdash \text{t1 = t2} & \quad \rightarrow \quad [ \ H \vdash \text{t1 =} \implies \text{t2} \\
& \quad ; \ H \vdash \text{t2 =} \implies \text{t1} ] \\
H \vdash \text{t1} \land \text{t2} & \quad \rightarrow \quad [ \ H \vdash \text{t1} \\
& \quad ; \ H \vdash \text{t2} ] \\
H \vdash \text{t1} \land \text{t2} \implies \text{tm} & \quad \rightarrow \quad [ \ H \vdash \text{t1} \implies (\text{t2 =} \implies \text{tm}) \\
& \quad ; \ H \vdash \text{t2} \implies (\text{t1 =} \implies \text{tm}) ] \\
H \vdash \text{t1} \land \text{t2} \implies \text{tm} & \quad \rightarrow \quad [ \ H \vdash \text{t1} \implies \text{tm} \\
& \quad ; \ H \vdash \text{t2} \implies \text{tm} ]
\end{align*}
\]

In addition, universal quantifiers are moved inwards as far as possible, and existential quantifiers in antecedents are eliminated. Several
theorems can be generated from this operation. If for example the antecedent is a conjunction of several parts, one theorem is generated for each permutation on the order of the parts.

The tactic tries to match each theorem \( \text{thm} \) in the list of “canonical” theorems with each assumption \( \text{asm} \) on the assumption list. Any theorem it can prove by applying \( \text{MATCH_MP thm asm} \) will once more be used to attempt a match with other assumptions. When no more matches are possible, the theorem is added to the assumption list so that \( \text{ASM_REWRITE_TAC} \) and \( \text{RES_TAC} \) have more to work with.

```
#LESS_CASES_IMP;;
|- !m n. "m < n \&\& (m = n) ==> n < m

#set_goal ["-(y < x)"; "0 < x"; "!z. -(z MOD x = x)",
"(y MOD x) < y")];
"(y MOD x) < y"
[ "!z. -(z MOD x = x)"
[ "0 < x"
[ "y < x"

() : void

#expand (IMP_RES_TAC LESS_CASES_IMP);;
OK.
"(y MOD x) < y"
[ "!z. -(z MOD x = x)"
[ "0 < x"
[ "y < x"
[ "-(y = x) ==> x < y"

() : void
```

Notice in the example that the theorem cannot be resolved with a quantified assumption. Also, there are two canonical theorems:

```
[ |- !m n. "m < n ==> (m = n) ==> n < m
 ; |- !m n. "(m = n) ==> m < n ==> n < m ],
```

but only the first finds a match with an assumption, so the other is discarded.

- \( \text{RES_TAC} \) operates only on the assumptions rather than a supplied theorem. It looks among the assumptions for candidate implications, and uses them just as \( \text{IMP_RES_TAC} \) does to “resolve” with other assumptions, adding the successful result of each resolution attempt to the assumption list. It will not use these added results to make
more matches, so sometimes the tactic will need more than a single application.

- Instead of matching the antecedents of an implicative theorem with assumptions, \texttt{MATCH_mp_TAC thm} matches the consequent of the implicative theorem \texttt{thm} with the goal, returning a new goal corresponding to the (matched) antecedent of \texttt{thm}. Any free variables thus introduced are existentially quantified, as in the following example using the built-in theorem \texttt{LESS_TRANS}.

```plaintext
#LESS_TRANS;;
| ! m n p. m < n \ n < p \implies m < p

#set_goal ("0 < n", "PRE n < n + n");
"(PRE n) < (n + n)"
[ "0 < n" ]

() : void

#expand (MATCH_mp_TAC LESS_TRANS);;
OK..
"?n'. (PRE n) < n' \ n' < (n + n)"
[ "0 < n" ]

() : void

#expand (EXISTS_TAC "n:znum");
OK..
"(PRE n) < n \ n < (n + n)"
[ "0 < n" ]

() : void
```

Tactics to finish off a proof
We have already mentioned \texttt{REFL_TAC}. A goal may also be solved in one step when we already have a theorem which matches the goal.

- \texttt{ACCEPT_TAC thm} solves a goal that is identical to \texttt{thm}
- \texttt{MATCH_ACCEPT_TAC thm} solves a goal that is an instantiation of \texttt{thm}

14.1.5 Tactics for substitution

- \texttt{SUBST_TAC [ thms ]} substitutes with a list of equational theorems throughout the goal.
CHAPTER 14. TACTICS AND TACTICALS

\begin{verbatim}
#g "((x > 0) => (x >= 0)) /\ \neg ((x = 0) = (x >= 0))";;
"(x > 0) => x >= 0) /\ \neg ((x = 0) = x > 0)"
()

#e (SUBST_TAC (map (SPECL ["x:num"; "0"])
  [ GREATER; GREATER_OR_EQ]));
OK.
"(0 < x => x > 0 \land (x = 0)) /\ \neg ((x = 0) = x > 0 \land (x = 0))"
()

SUBST_OCCS_TAC may be used to pick out which occurrences of a term
\texttt{lhs} to replace in a goal. It takes theorem in the form of an equation,
e.g. \texttt{lhs = rhs}, and an integer list which acts as keys stating which
of the occurrences of \texttt{lhs} to replace by \texttt{rhs} in the goal.

\begin{verbatim}
#g "((x > 0) => (x >= 0)) /\ \neg ((x = 0) = (x >= 0))";;
"(x > 0) => x >= 0) /\ \neg ((x = 0) = x > 0)"
()

#e (SUBST_OCCS_TAC [ ([1; 2], SPECL ["x:num"; "0"] GREATER);
  ([2], SPECL ["x:num"; "0"] GREATER OR_EQ)]);
OK.
"(0 < x => x >= 0) /\ \neg ((x = 0) = x > 0 \land (x = 0))"
()

\end{verbatim}

\begin{itemize}
  \item SUBST_TAC thm substitutes with a single theorem in the goal.
  \item SUBST_ALL_TAC thm substitutes with a single theorem both in the
    conclusion and all the assumptions of the goal. It is occasionally very
\end{itemize}
useful, for example, should we wish to change 1 to \texttt{SUC 0} everywhere prior to rewriting from the assumptions.

\begin{verbatim}
#set_goal ( [ "x > 1" ; "y > 1" ], "(x * y) > 1" );
"(x * y) > 1"
[ "y > 1" ]
[ "x > 1" ]

() : void

#e(SUBST_ALL_TAC (num_CONV "1"));
OK.
"(x * y) > (SUC 0)"
[ "y > (SUC 0)"
[ "x > (SUC 0)"

() : void
\end{verbatim}

14.1.6 Tactics for rewriting

Since rewrite rules and rewrite tactics work in the same way, you are referred to chapter 13 for the details of rewriting. We remind you that the free variables in rewriting equations are all fully generalised before any rewriting is done, and that rewriting is carried out on all the subterms of the theorem to be rewritten, and repeatedly until no rewrite applies. When divergence would occur, revert to \texttt{ONCE_REWRITE_TAC} or use substitution.

Here are the various forms of rewriting tactic which prefix the stem \texttt{REWRITE_TAC} by one or more of

[ \texttt{FILTER}_-], [ \texttt{PURE}_-], [ \texttt{ONCE}_-], [ \texttt{ASM}_-]

They have the following interpretations:

- \texttt{REWRITE_TAC [ arg]} rewrites with the arguments and with a built-in list of theorems (see Appendix E).
- \texttt{ASM} adds the theorems on the assumption list to the list of rewriting agents.
- \texttt{ONCE} navigates a the subterms in a goal from top to bottom moving on to the next subterm after one successful rewrite.
- \texttt{PURE} does not use the built-in list of theorems (nor the assumptions unless combined with the stem \texttt{ASM} It only rewrites with the supplied argument list.
• **FILTER** takes a predicate through which it filters all the rewriting agents. It makes sense to use **FILTER** only in conjunction with **ASM** since there is no point in supplying an explicit list of arguments and then filtering them.

### 14.1.7 Miscellaneous tactics

- **ALL_TAC** is a tactic that never fails, and does nothing to the goal. It is often used to represent an empty leg in a THENL list.
- **NO_TAC** is a tactic that always fails. It is most used in defining tacticals.
- **FAIL_TAC** is a tactic that also fails, with the supplied failure string. It can be useful in defining tacticals which give useful information for tracing where a tactical fails.

### 14.2 Tacticals

Compound tactics may be constructed from smaller ones using *tacticals*. Here are the most important tacticals:

1. **THEN**: tactic -> tactic -> tactic. **THEN** is an ML infix whose definition is rather convoluted and is not given here. If **Tac1** and **Tac2** are tactics, then **Tac1 THEN Tac2** evaluates to a tactic that first tries **Tac1**. If this fails, the compound tactic fails. If not, it will try **Tac2** on each subgoal produced by the application of **Tac1**.

2. **ORELSE**: tactic -> tactic -> tactic. **ORELSE** is an ML infix defined by

\[
\text{let } (\text{Tac1 ORELSE Tac2}) \ g = \text{Tac1} \ g \ ? \text{Tac2} \ g \ ;;
\]

If **Tac1** and **Tac2** are tactics, then **Tac1 ORELSE Tac2** evaluates to a tactic that first tries **Tac1**. It will try **Tac2** only if **Tac1** fails.

3. **REPEAT**: tactic -> tactic -> tactic is an ML function defined by

\[
\text{let REPEAT Tac g =}
(\text{((Tac THEN (REPEAT Tac)) } \ ? \text{ALL_TAC}) \ g) \ ;;
\]

which repeatedly applies **Tac** to the goal **g** until it fails.
4. **THENL**: tactic -> tactic list -> tactic. If Tac produces n subgoals (e.g., BOOL_CASES_TAC produces two), and Tac1, Tac2, ..., Tacn are tactics, then TAC THENL [ Tac1; Tac2; ... ; Tacn ] is a tactic that first applies TAC to the current goal, and then applies Tack to the kth subgoal produced by the application of Tac.

### 14.2.1 List tacticals

We have already met and used a few of the list tacticals, for example MAP_EVERY and ASSUM_LIST. The most commonly used are:

- **EVERY** [ tac1; tac2; ...; tacn ] is a neat way of writing
  
  tac1 THEN tac2 THEN ... THEN tacn

- **MAP_EVERY** tac L is simply defined as EVERY(map tac L). We have used it often to map the tactic BOOL_CASES_TAC over a list of terms.

- **ASSUM_LIST** was shown in action in the previous chapter. It has a simple definition

```plaintext
#let ASSUM_LIST f (asl, concl) = f (map ASSUME asl) (asl, concl);
ASSUM_LIST = - : (thm list -> (term list # *) -> * *) -> (term list # *) -> * *
```

When handed as an argument to expand, it takes a tactic f and the goal (pattern matched to asl, concl)) as arguments. It copies the assumption list by mapping ASSUME down it and then hands the copy and the goal over to the tactic f.

- **EVERY_ASSUM** is a derivative of ASSUM_LIST defined by

```plaintext
#let EVERY_ASSUM = ASSUM_LIST o MAP_EVERY;
EVERY_ASSUM = - : (thm tactic -> tactic)
```

Suppose that the goal's assumption list is [ tm1; tm2; ...; tmn]. Then EVERY_ASSUM tac is equivalent to

```
tac tm1 THEN tac tm2 THEN ... THEN tac tmn
```

A typical example is
CHAPTER 14. TACTICS AND TACTICALS

```
#set_goal([ "0 = x"; "2 = y" ], "x < ( y + z)");
"x < (y + z)"
  
  [ "2 = y" ]
  [ "0 = x" ]

( ) : void

#e(EVERY_ASSUM (SUBST1_TAC o SYM));;
OK..
"0 < (2 + z)"
  
  [ "2 = y" ]
  [ "0 = x" ]

( ) : void
```
Chapter 15

Theorem continuations

In large and complex HOL proofs, it is quite usual for us to build up a number of assumptions on the assumption list. Often these assumptions are not quite what we want but are put on the assumption list for further manipulation (for example, by IMP_RES_TAC or RES_TAC). A common side effect is that several other theorems that we never need also get added to the assumption list. Again we frequently use an assumption on the assumption list once but leave it there until the proof is completed. Thus there is a tendency for assumption lists to accumulate assumptions that we do not need any more. Is there anything we can do about this?

The obvious way is to intercept and manipulate assumptions and use them without adding them to the assumption list at all, or if that is unwieldy, then at least to manipulate them into a better shape before putting them onto the assumption list. Theorem continuations take one or more theorem tactics as arguments. They perform operations which generate assumptions, but rather than adding these to the assumption list, they instead pass ASSUME’d versions to the theorem tactic arguments.

We divide theorem continuations into two broad categories: ones that generate theorem/assumptions, and ones which manipulate given theorem/assumptions. We shall refer to the theorem/assumptions as theorems throughout this chapter, since they are indeed cast as theorems by the tactics.

15.1 Theorem Generators

The main sources for theorems in this style of proof are the antecedents of implications, resolvents from the assumption list, assumptions from induction proofs, and the direct use of theorems.

15.1.1 DISCH_THEN: working with implications

So far, when we have had a goal of the form \((x = \text{rhs}) \implies G(x)\), we have pushed the antecedent onto the assumption list and then rewritten from the assumptions.
CHAPTER 15. THEOREM CONTINUATIONS

If we only need the rewrite (or equivalently in this case, substitution) provided by the antecedent once, we can use it and throw it away using the theorem continuation DISCH_THEN.

Given a goal of the form \((asl, a => b)\), the theorem continuation DISCH_THEN \(tac\) works as follows.

1. **DISCH_THEN** takes the conclusion of the goal and breaks it into two parts, the antecedent \(a\) and a consequent \(b\).

2. The antecedent is taken as a theorem (. \(\vdash a\)) by applying **ASSUME** to it, which is how it would be understood should we retrieve it after pushing it onto the assumption list with **DISCH_TAC**.

3. It then applies the theorem tactic \(tac\) to (. \(\vdash a\)), producing a tactic which is in turn applied to the goal (asl, \(b\)).

Note that nothing is added to the assumption list.

Since we have a handle on the antecedent in step 3 above, we have the opportunity to transform it should it not be in the right shape. We can, for example, also apply a forward inference rule \(f\) to the theorem by giving
the argument to DISCH_THEN as (tac o f). This will apply
tac(f( |- a)) to the goal (asl, b). In this next example we present a
multistep forward proof to turn a negated expression into an equation which
can be used for substitution. The example is admittedly not the most direct
proof of the result, but demonstrates the power of the technique.

```
#i "(~0 < x) ==> (x * y = 0) \ (x + y = y)";
"0 < x ==> (x * y = 0) \ (x + y = y)"
()
```

From the antecedent we would like to derive that \(x = 0\). The forward proof
is shown below, step by step. We begin with one derived\(^1\) and two built-in
thms used in the proof.

```
#th1,NOT_LESS_O,NOT_CLAUSES;;
(l |- "m < n ==> ~(m = n) ==> n < m,
 l |- ~(n < 0,
 l |- (~(t, ~(t = t) \ (T = F) \ (F = T)))
 : (thm # thm # thm))
```

```
#MATCH_MP th1 (ASSUME ~(O < x));
. l |- ~(O = x) ==> x < 0
```

```
#PURE_ONCE_REWRITE_RULE [NOT_LESS_O] it;;
. l |- ~(O = x) ==> F
```

```
#NOT_INTHO it;;
. l |- ~(O = x)
```

```
#PURE_ONCE_REWRITE_RULE [NOT_CLAUSES] it;;
. l |- O = x
```

```
#SYM it;;
. l |- x = 0
```

The forward proof performs modus ponens with theorem th1 and the an-
tecedent cast as a theorem, rewrites the result to replace the consequent
of the resulting implication with F, transform the \(\ldots \Rightarrow F\) part of the
theorem to a negation of the antecedent, eliminates the double negation
by a rewrite, and then SYM's the result to get the conclusion in the form
needed for substitution. The individual steps are shown in the following. To
use this forward proof combined with the theorem continuation, we simply
compose the proof steps as follows.

\(^1\)The forward proof of this theorem is let th1 = hd (IMP_CANON_LESS_CASES_IMP);;

15.1. THEOREM GENERATORS

351
15.1.2 Resolvants

Resolution can generate assumptions either from the existing assumptions, or using a provided theorem which is resolved with the assumptions. We begin with **IMP_RES_THEN** which is the simplest in operation.

**IMP_RES_THEN**

The theorem argument to **IMP_RES_THEN** must be an implication, or a theorem from which an implication can be derived by the transformations described for **IMP_RES_TAC** on page 341. The following example will use the resolvent as a rewriting agent, and for ease of composing the tactic we define a rewriting theorem tactic that takes a single theorem argument.

There are a few subtleties in this tactic which may not be obvious. First is the use of rewriting instead of substitution. Notice that the theorem **INV_PRE_LESS_EQ** has the variable \( m \) quantified in the consequent. This
variable will be quantified in the resolvants as well. One possibility would be to specialize it to the variable $s$ as follows.

```haskell
#b();
"(PRE s) <= (PRE t) ==> s > t"
[ "0 < t" ]
()

#e (IMP_RES_THEN (SUBST_TAC o (SPEC "s: num")) INV_PRE_LESS_EQ);;
OK.
evaluation failed SUBST_TAC -- dest_eq
```

Is this failure surprising? Understanding why it occurs is important if you want to appreciate the way resolution is implemented. When the theorem argument is handed to `IMP_RES_THEN` it is first made into a list of canonical forms of the theorem. We can look at this list by applying `RES_CANON` directly to the theorem.

```haskell
#RES_CANON INV_PRE_LESS_EQ;
[!n. 0 < n ==> (!m. (PRE m) <= (PRE n) = m <= n);
!n. 0 < n ==> (!m. (PRE m) <= (PRE n) = m <= n);
!n. 0 < n ==> (!m. m <= n ==> (PRE m) <= (PRE n))]
: the list

#e (IMP_RES_THEN ASSUME_TAC INV_PRE_LESS_EQ);;
OK.
"(PRE s) <= (PRE t) ==> s > t"
[ "0 < t" ]
[ "!m. (PRE m) <= (PRE t) = m <= t" ]
[ "!m. (PRE m) <= (PRE t) = m <= t" ]
[ "!m. m <= t ==> (PRE m) <= (PRE t)" ]
()

: void
```

There is the answer. The equality of the consequent is split into two implicative forms as well. Thus the theorems generated by the resolution include ones which are not equations, and `SUBST_TAC` fails when it tries to take apart these equations. Notice that despite generating a list of resolvants, the theorem tactic argument takes a single theorem argument. This tactic is applied to each resolvant in turn, attempting 3 substitutions in some order. For this particular example the original tactic using rewriting is the most effective.

The next step is considering the problem of more than one antecedent. The relevant theorem and the new goal follow.

---

2 The theorem `INV_PRE_LESS` has changed since this was written, removing the redundant antecedent $0 < n$. All this needs revision.
Twice as many canonical forms are produced, caused by the two ways of ordering the antecedent conjuncts.

The important difference between \texttt{IMP\_RES\_THEN} and \texttt{IMP\_RES\_TAC} that we have seen earlier is that the former only tries to achieve every single match a single antecedent with each canonical theorem, whereas the latter continues trying to match the resolvants with other assumptions. Thus we need to perform two applications of \texttt{IMP\_RES\_THEN} to get a workable result.

This has done what we wanted, but some things going on behind the scenes may be surprising. Let us back up a bit and look at the resolvants produced by using \texttt{ASSUME\_TAC} instead of rewriting.
This is not perhaps what you would expect, but is not unpredictable. The first resolution matches each of the two assumptions with every canonical form, producing 12 resolvents. The second resolution again matches each of the two assumptions with every one of the 12 resolvents, but in addition generates new canonical forms from the resolvents that have an equation in the consequent, and matching each of these also with each assumption generates another 16 (duplicate) resolvents, giving us 40 in total! The lesson to be observed is that resolution can be a very expensive operation, and care should be taken to limit possible matches where problems may result. Here again the techniques to constrain the matches are not obvious. Specializing the theorem to the desired values for example will not work, since they will be generalized again when put into canonical form. One of the best (but surprisingly little used) ways of achieving this is to specialize and then "freeze" the theorem so the variables will not be generalized.

```plaintext
let spec_thm = SPEC "s:num" INV_PRE_LESS;;
spec_thm = |- 0 < s ==> (!n. (PRE s) < (PRE n) = s < n)

let (FREEZE_THEN (IMP_RESTHEN_ASSUME_TAC spec_thm));;
ok..
"(PRE s) < (PRE t) ==> ~s >= t"
[ "0 < s" ]
[ "0 < t" ]
[ "!n. (PRE t) < (PRE n) = t < n" ]
[ "!n. (PRE s) < (PRE n) = s < n" ]
[ "!n. (PRE t) < (PRE n) ==> t < n" ]
[ "!n. (PRE s) < (PRE n) ==> s < n" ]
[ "!n. t < n ==> (PRE t) < (PRE n)" ]
[ "!n. s < n ==> (PRE s) < (PRE n)" ]
[ "!n. s < n ==> (PRE s) < (PRE n)" ]

() : void
```
**THEOREM CONTINUATIONS**

**FREEZE** is another theorem continuation that takes a theorem tactic argument and a theorem. It prevents generalizing the variables of the theorem by adding a copy of the theorem conclusion to its hypotheses. Because the variables are then not free in the hypotheses of the theorem, they cannot be generalized during resolution. Notice that the additional implications are still generated. This is intrinsic in the definition of resolution, and is not readily avoided.

The explicit programming of two resolutions is also awkward, particularly if several matches must be made. One approach is to recursively define a theorem continuation which applies itself a given number of times. Such a specialized form is defined below.

```plaintext
#let rec IMP_RES_n_THEN n (ttac:thm_tactic) thm =
  (n = 0) => ttac thm
  | (TRY o ((IMP_RES_THEN (IMP_RES_n_THEN (n - 1) ttac))) thm);
IMP_RES_n_THEN = := (int -> thm_tactical)
```

This continuation applied to a number \( n \) will attempt resolution \( n \) times. The **TRY** operator prevents the entire tactic from failing if one of the solvers fails to find a match before all the \( n \) resolutions are accomplished.

```plaintext
#b();
"(PRE s) < (PRE t) ==> s >= t"
  [ "0 < s" ]
  [ "0 < t" ]
  [ "!n. (PRE t) < (PRE n) = t < s" ]
  [ "!n. (PRE s) < (PRE n) = s < n" ]
  [ "!n. (PRE t) < (PRE n) ==> t < n" ]
  [ "!n. (PRE s) < (PRE n) ==> s < n" ]
  [ "!n. t < n ==> (PRE t) < (PRE n)*" ]
  [ "!n. s < n ==> (PRE s) < (PRE n)*" ]
( ) : void
```

```plaintext
#c (FREEZE_THEN (IMP_RES_n_THEN 2 ASSUME_TAC) spec_thm);
OK...
"(PRE s) < (PRE t) ==> s >= t"
  [ "0 < s" ]
  [ "0 < t" ]
  [ "!n. (PRE t) < (PRE n) = t < s" ]
  [ "!n. (PRE s) < (PRE n) = s < n" ]
  [ "!n. (PRE t) < (PRE n) ==> t < n" ]
  [ "!n. (PRE s) < (PRE n) ==> s < n" ]
  [ "!n. t < n ==> (PRE t) < (PRE n)*" ]
  [ "!n. s < n ==> (PRE s) < (PRE n)*" ]
( ) : void
```
There are two repeat constructs for theorem continuations, called `REPEAT TCL` and `REPEAT_GTCL`. The first is repeatedly applied to the theorem argument until it fails, while the latter is applied repeatedly to the goal until it fails. Since we are resolving repeatedly with the goal assumptions, the latter is what is needed in this case; the other form is described later. The same number of matches will result as from explicitly programming two resolution steps in this case, so “freezing” the theorem is again recommended.

```tcl
#b() ;;
"(PRE a) < (PRE t) ==> "s >= t"
  [ "0 < a" ]
  [ "0 < t" ]
  [ "!n. (PRE t) < (PRE n) = t < n" ]
  [ "!n. (PRE a) < (PRE n) = s < n" ]
  [ "!n. (PRE t) < (PRE n) ==> t < n" ]
  [ "!n. (PRE a) < (PRE n) ==> s < n" ]
  [ "!n. t < n ==> (PRE t) < (PRE n)" ]
  [ "!n. s < n ==> (PRE a) < (PRE n)" ]

() : void
#e (FREEZE_THEN (REPEAT_GTCL IMP_RES_THEN_ASSUME_TAC) spec_thm) ;;
OK...
"(PRE a) < (PRE t) ==> "s >= t"
  [ "0 < a" ]
  [ "0 < t" ]
  [ "!n. (PRE t) < (PRE n) = t < n" ]
  [ "!n. (PRE a) < (PRE n) = s < n" ]
  [ "!n. (PRE t) < (PRE n) ==> t < n" ]
  [ "!n. (PRE a) < (PRE n) ==> s < n" ]
  [ "!n. t < n ==> (PRE t) < (PRE n)" ]
  [ "!n. s < n ==> (PRE a) < (PRE n)" ]
  [ "!n. s < n ==> (PRE s) < (PRE n)" ]

() : void
```

**RES_TAC**

`RES_THEN` performs resolution between implicative assumptions and other assumptions, handing the resulting theorems on the the theorem tactic argument. The example sets up a goal, and then uses resolution to get to a state where we can resolve two assumptions.
The first and last assumptions can be resolved to derive the equality of \( k \) and \( \text{k MOD n} \). This result is substituted directly into the goal.

\[ \text{ANTE_RES_THEN} \]

A theorem continuation for resolution without a corresponding tactic is \text{ANTE_RES_THEN}, which instead of taking an implicative theorem, takes a theorem which is matched to the antecedent of an implicative assumption. We set up the previous subgoal again.
This looks somewhat complicated, but if you think about the last steps is should be quite clear. \texttt{IMP\_RES\_THEN} resolves the theorem \texttt{OR\_LESS} with the assumption \texttt{"(SUC k) \leq n"} giving the theorem (\( \neg k < n \)). This is handed to \texttt{ANTE\_RES\_THEN}, which resolves it with the only implicative assumption to produce the theorem (\( \neg \, k \mod n = k \)). This is handed to \texttt{SUBST\_TAC} to substitute for the left hand side term in the goal.

\section*{15.2 Manipulating Theorems}

The previous chapter (14) has presented a number of ways of applying either theorems or assumptions in a tactical proof. Any tactic which takes a theorem as an argument and returns a tactic is called a \texttt{thm\_tactic}. Obvious examples are \texttt{SUBST\_TAC}, \texttt{ASSUME\_TAC}, \texttt{ASM\_CASES\_TAC}, \texttt{IMP\_RES\_TAC}, and \texttt{DISJ\_CASES\_TAC}. All of these may be used with theorem continuations.

We have looked at continuations that primarily generate theorems, and now we turn our attention to ones which manipulate these in some way that is not readily possible using the forward proof techniques already given. These theorem continuations generally are applicable to assumptions with a particular form. Throughout this section we shall use examples which have an implicative goal, so a theorem of the appropriate form is obtained by applying \texttt{DISCH\_THEN}. It should be obvious that any source of obtaining a theorem of the appropriate form, such as resolution or indeed by using an available theorem, is equally applicable to the techniques given.

\subsection*{15.2.1 Existential quantifiers}

Eliminating existential quantifiers in tactical proofs is relatively simple, especially when compared to eliminating them in forward proofs. If we have an existentially quantified antecedent, we can “choose” a variable of the same name as a “witness” for the body of the expression, and use the resulting theorem for, in the following example, substituting in the goal.
If, for some reason, you wish to determine what variable is substituted for the quantified variable, you may select this using \texttt{\texttt{X\_CHOOSE\_THEN}} instead.

The variable chosen (and it must be a variable, not any other term) must not be free both in the body of the quantified expression, nor in the conclusion of the goal.

15.2.2 Conjunctions

Should the antecedent of a goal be a conjunction of terms of the form \texttt{lhs = rhs} for example

\[ (x = 0) \land (y = 1) \implies (x * y = 0) \land (x + y = 1) \]

and we want to substitute for the left hand side of each term into the goal, we need to split the antecedent into its constituents before we can apply the individual substitutions. The first step is to apply \texttt{DISCH\_THEN} which (in this case) hands over \texttt{|- (x = 0) \land (y = 0)} and a modified goal to its argument. We now introduce \texttt{CONJUNCTS\_THEN} which has type \texttt{(thm -> tactic) -> (thm -> tactic)}, or \texttt{thm\_tactical} for short.
CONJUNCTS\_THEN\_tac takes a theorem of the form A |- p \(\land\) q and turns it into tac (A |- p) THEN tac (A |- q). In the case above, DISCH\_THEN transforms the goal into

\[(\Box, "(x \ast y = 0) \land (x + y = 1)")\]

and hands over 1- (x = 0) \(\land\) (y = 1) to CONJUNCTS\_THEN\_SUBST1\_TAC. SUBST1\_TAC (1- x = 0) THEN SUBST1\_TAC (1- y = 1) is then applied to the transformed goal. Perhaps this is easier to follow as a trace which shows the individual steps undergone in applying the tactic to it and the end goal achieved.

\[
\begin{align*}
 & (\text{DISCH\_THEN (CONJUNCTS\_THEN\_SUBST1\_TAC)}) \\
 & (\Box, (x = 0 \land y = 1) \implies (x \ast y = 0 \land x + y = 1)) \\
 & \implies (\text{CONJUNCTS\_THEN\_SUBST1\_TAC (1- x = 0 \land y = 1)}) \\
 & (\Box, (x \ast y = 0 \land x + y = 1)) \\
 & \implies (\text{SUBST1\_TAC (1- x = 0) THEN SUBST1\_TAC (1- y = 1)}) \\
 & (\Box, (x \ast y = 0 \land x + y = 1)) \\
 & \implies (\text{SUBST1\_TAC (1- y = 1)}) \\
 & (\Box, (0 \ast y = 0 \land 0 + y = 1)) \\
 & \implies (\Box, (0 \ast 1 = 0 \land 0 + 1 = 1))
\end{align*}
\]

In the case of an antecedent with more than two conjunctions we have to apply CONJUNCTS\_THEN repeatedly. In such cases we apply the theorem continuation REPEAT\_TCL to CONJUNCTS\_THEN. Yes THEN\_TCL, ORELSE\_TCL, ALL\_TCL, and NO\_TCL are also built-in.
It can sometimes be the case that different uses will be made of the two conjuncts. In this case `CONJUNCTS_THEN2` can be used to apply different tactics to each conjunct one after the other.

15.2.3 Disjunctions

When the conclusion of the goal is an implication and the antecedent is a disjunction, say \( a \lor b \Rightarrow G \), we can split it into two subgoals with `DISJ_CASES_THEN` as below.
DISJ_CASES_THEN accepts the theorem \( \vdash (x = 0) \lor (y = 0) \Longrightarrow (x \cdot y = 0) \) and the tactic SUBST1_TAC. It then applies SUBST1_TAC (. \( \vdash x = 0 \)) to the goal (\( \Box \), \( x \cdot y = 0 \)) producing the subgoal (\( \Box \), \( x \cdot 0 = 0 \)). It also applies SUBST1_TAC (. \( \vdash y = 0 \)) to the goal (\( \Box \), \( x \cdot y = 0 \)) producing the second subgoal (\( \Box \), \( 0 \cdot y = 0 \)).

Here is a trace of the call:

\[
\begin{align*}
\text{e(DISCH_THEN (DISJ_CASES_THEN SUBST1_TAC))} \\
\quad \rightarrow \text{(DISCH_THEN (DISJ_CASES_THEN SUBST1_TAC))} \\
\quad \rightarrow (\Box, (x = 0 \lor y = 0) \Longrightarrow (x \cdot y = 0)) \\
\quad \rightarrow (\Box, x \cdot y = 0) (\vdash (x = 0) \lor y = 0) \\
\end{align*}
\]

Two subgoals are generated:

Case 1: \( \text{SUBST1_TAC (} \vdash x = 0) (\Box, x \cdot y = 0) \) 
\( \rightarrow (\Box, 0 \cdot y = 0) \)

Case 2: \( \text{SUBST1_TAC (} \vdash y = 0) (\Box, x \cdot y = 0) \) 
\( \rightarrow (\Box, x \cdot 0 = 0) \)

As with the continuations for conjuncts, we can also use different theorem tactics for each disjunct.
The first disjunct is substituted directly, while the existential quantifier is first eliminated from the second before substitution. For theorems with more than 2 disjuncts, you may use \texttt{DISJ\_CASES\_THENL}, which takes a list of theorem tactics, one to be used for each disjunct.

\subsection*{15.2.4 Combinations and Permutations}

A more powerful theorem continuation that will account for conjunctions, disjunctions, and existentially quantified theorems is \texttt{STRIP\_THM\_THEN}. It is the common work horse but is occasionally too powerful, so you need to know the simpler cases above. The repetition (supplied by \texttt{REPEAT\_TCL}) is need to break the conjuncts into three.

Each application of \texttt{STRIP\_THM\_THEN} breaks up one conjunct, disjunct, or eliminates one existential quantifier.

\section*{15.3 General Induction}

We treat general induction as a special case. We have already seen the induction tactics which are particular to the datatypes \texttt{num} and \texttt{list}. Both of
these are instances of the more general theorem continuation `INDUCT_THEN`. We may in fact perform structural induction on any concrete datatype. The arguments are a structural induction theorem of the form returned by the `prove_induction_thm`, usually derived when the type is defined and a `thm_tactic`. The induction theorems for types `num` and `list` follow.

```
#INDUCTION;;
|- !P. P 0 /
    (!n. P n ==> P(SUC n)) ==> (!n. P n)

#list_INDUCT;;
|- !P. P [] /
    (!t. P t ==> (!h. P(CONS h t)) ==> (!l. P l))
```

Simplified definitions of the corresponding built-in tactics follow.

```
let INDUCT_TAC =
    let tac = INDUCT_THEN INDUCTION_ASSUME_TAC in
    \g. tac g ? failwith 'INDUCT_TAC';;

let LIST_INDUCT_TAC =
    INDUCT_THEN list_INDUCT_ASSUME_TAC;;
```

Any inductive assumption is handed to the theorem tactic, which in these cases just puts them on the assumption list. More direct uses of the assumptions are possible using the techniques already presented in this chapter.

## 15.4 Proof Hacking

A warning should accompany this discussion, which has delved into some fairly obscure ways of doing tactical proofs. We have attempted to extend the range of proof techniques that are available to you. What has been missed is establishing when one or another approach is preferable. To some extent this is guided by your own familiarity and understanding of the techniques, as you will tend to mainly use the same techniques over and over again. Only when some particular difficult problem in a proof arises will new techniques be worth exploring.

There are however some fairly objective (if not readily quantifiable) measures which can be applied to judging appropriateness of selected approaches. Aside from succeeding in proving a goal which can be measured, one would hope that a technique would be robust to possible future changes in the underlying system. We have emphasized this earlier when discussing the use of assumptions without relying on their ordering.

We consider another measure by far the most important. This is the legibility of the proof, both to other readers and indeed to yourself some
months after completing the proof. Legibility of a tactical proof, even for experienced HOL users, can be impossible to achieve in many cases, but it can often be improved enormously if some thought is given to the issue at the time. Resolution involving the matching of several antecedents of several possibilities can be difficult, as can deeply nested applications of a mixture of forward and tactical proofs. Several things can be done to improve the situation. The use of \texttt{let} bindings to keep track of terms or theorems can help. Certainly formatting of the tactic can isolate the extent of complex proof steps in, for example, the theorem tactic argument to theorem continuations. The sequence of steps can also be highlighted by the format. The most illuminating device is documentation. Annotations identifying the effect of particular steps, the relevance of particular terms, the effect of sequences of steps, and so on can provide a higher level view of what is going on in the proof. This may not be required when the proof is only a couple of lines long (although we can think of examples where a note or two would have been very helpful), but when a proof extends for pages, it will not be very readable.

One must really acknowledge that economy of proof steps should also be considered, but not usually as the primary measure. It is often the case that the best proof by any other measure one may choose is going to be the most efficient. If, however, you check back the proofs using \texttt{IMP\_RES\_THEN} you may find this is not the case. This chapter then has illustrated both some rather obscure proof techniques, and some very powerful tools which you can add to your repertoire, and use for your pleasure. Please though keep in mind the notion of legibility, and try to apply these proof techniques in ways which can clarify rather than obscure your proofs.
Part VI

Clocked sequential circuits
Chapter 16

Basic techniques

Clocked sequential circuits are networks of combinational circuits separated by registers. The combinational circuits provide the functionality of the network, the registers the stability. Figure 16.1 sketches a design with four combinational circuits $C_k$ each fed by a separate register $R_k$, $k = 1, 2, 3, 4$. Informally, the functionality of the circuit is $O(t+n) = C_4(C_3(C_2(C_1 I)))$ or $O(t+n) = (C_4 \circ C_3 \circ C_2 \circ C_1) I$ where the delay of the circuit is $n$.

Figure 16.1 Sequential sub-net

Suppose at any given time $t$, the input $I$ and the registers are stable. Some time later the combinational circuits they feed will have computed and settled, and the next time a register is docked, it will read the (stable) value computed by the combinational circuit that feeds it. Care has to be taken that computed values merely feed into the next register downstream and cannot streak through several registers. One way of ensuring this is to adopt the two-phase clocking strategy in which the odd-numbered registers are clocked on clock $\phi_o$ and the even-numbered registers are clocked on clock $\phi_e$. The two clocks are non-overlapping (never high at the same time and sufficiently well-separated).\(^1\)

We are NOT interested in values on lines at all actual times, only in the stabilised values that will be stored in registers. We model combinational networks by assuming that they fire instantaneously, and put all the delay

\(^1\)We do not have time to formalise clocking strategies in this book. Read about two-phase clocking in Mead and Conway’s classic text [75, page 65 et seq.]. Dhingra [34, 35] describes the formalisation of a more advanced clocking strategy in HOL.
into the registers. We then sample the circuits at times when the registers have settled. This enables us to model the passage of time by regular ticks at times 0, 1, 2, ... where we assume that all combinational sub-circuits have time to settle between clock ticks.

In general, then, we express the signal value on line p at time t by pt, where p is a function from num→bool (p takes a time index of type num and maps it into a bool value).

Our specifications have to express how signals on lines vary with the time index. Typical relations in a specification are

\[ z(t) = \text{expr}(t) \quad \text{for combinational circuits, and} \]
\[ z(t+n) = \text{expr}(t) \quad \text{for a register with delay n} \]

As examples here are the (slightly simplified) specifications of a four input mux gate

\[
\text{mux41_spec sel1 sel2 i0 i1 i2 i3 (z::num->bool) = !t::num.}
\]
\[
z(t) = \begin{cases} 
\text{\neg sel1 t \lor \neg sel2 t} & \Rightarrow i0 t \\
\text{\neg sel1 t \lor sel2 t} & \Rightarrow i1 t \\
\text{sel1 t \lor \neg sel2 t} & \Rightarrow i2 t \\
\text{i3 t} & 
\end{cases}
\]

and a clocked 1-bit delay

\[
\text{FD1 clk (i::num->bool) q qn = !t .}
\]
\[
(q(t+1)) = \begin{cases} 
\text{\neg(clk t \Rightarrow i t | q t)} & \lor \\
\text{(not(clk t \Rightarrow i t | q t))} & 
\end{cases}
\]

Just as we did for combinational circuits, we make every effort to find a generic specification style suitable for a range of sequential circuits. In particular, we would like the specification of a row of boxes to have the same form as that of one box. A suitable abstraction is nEQL and we find that we can specify a line of FD1 is by

\[
\text{nFD1 clk (i::num->num->bool) q qn = !t .}
\]
\[
(nEQL (q(t+1))) (clk t \Rightarrow i t | q t) n) \lor \\
(nEQL (qn(t+1))) (\neg(clk t \Rightarrow i t | q t)) n)
\]

In this chapter, we run through some through some simple verifications which are rich enough to display all most of the standard problems and how to overcome them. Of particular importance are:

1. abstracting from bits to buses (using nEQL)
2. conversions for moving and cancelling \( \forall s \)

3. dealing with hidden lines.
   Since signals flowing along hidden lines are functions of time, we have to match the signal flowing out from one circuit with a signal flowing into another over all time indices. This turns out to be quite simple using standard \( \lambda \) conversion techniques.

4. using quantification to hide unused lines.
   Although most registers provide both \( q \) and \( q_n \) lines to show the current state and its inverse, many applications only require \( q \). What do we do with the unused \( q_n \)?

Chapters 17 is a set of case studies dealing with counters and registers. Chapter 18 is a large case study of finite state machines.

### 16.1 Library circuits

**FD1 — a primitive D-type flip flop.** The simplest clocked circuit in our sequential catalogue is a D-type flip flop (see figure 16.2) which, when regularly clocked, delays the incoming signal by one time unit.

![1-bit register](image)

Figure 16.2 1-bit register

One possible specification is

```plaintext
FD1 clk (i: num -> bool) q qn
= ! t : num.
  (q (t+1) = (clk t => i t | q t)) /\  
  (q_n(t+1) = '(clk t => i t | q t))
```

The outputs from this D-type are undefined at time 0. When clocked, the output on \( q \) echoes the previous input value, and the output on \( q_n \)
inverts the previous input value. I.e., when continuously docked, if the input sequence on \( i \) starting at time 0 is \( i_0, i_1, i_2, \ldots \) then the output sequence on \( q \) will be \( \text{unknown} \), \( i_0, i_1, \ldots \), and the output sequence on \( qn \) will be \( \text{unknown} \), \( \sim i_0, \sim i_1, \ldots \). When not clocked the device maintains its state.

We write the defining relations in the form \( z(t+1) = \text{rhs} \ t \) rather than \( z(t) = \text{rhs}(t-1) \) to make it quite clear clear that \( z(0) \) is unspecified. (We would also have to take extra care at time 0 since \( \text{rhs}(0-1) = \text{rhs} \ 0 \) in HOL.) Should we wish to specify an initialised device, we simply append extra constraints, as with

\[
\begin{align*}
\text{FD1_2} &::= ! t:\text{num}. \\
& \quad (q(t+1) = (\text{clk} t => i \ t \mid q \ t) \land (q 0 = F)) \land \\
& \quad (qn(t+1) = \sim (\text{clk} t => i \ t \mid q \ t) \land (qn 0 = T))
\end{align*}
\]

As the final touch, since the values on \( q \) and \( qn \) shadow each other, we adopt the \texttt{let} style of definition which both expresses the calculation once and textually emphasizes this fact. Of course, we will be able to unfold any \texttt{let} definitions in actual theorems by applying \texttt{let\_RULE} and in actual goals by expanding with \texttt{let\_TAC} (see chapter 14).

```
#let FD1 = new_definition
('FD1',
  "FD1 clk (i:num->bool) q qn
  = ! t:num. in
    ( q (t+1) = res)
    /
    (qn(t+1) = ~res)
  );;
FD1 = ...
```

\textbf{Aside}: without the extra brackets after the \texttt{in}, HOL parses the body of this definition as

\[
\begin{align*}
\text{let res} &::= (\text{clk} t => i \ t \mid q \ t) \text{ in } q(t+1) = \text{res} \\
& \quad \land \\
& \quad qn(t+1) = \sim \text{res}
\end{align*}
\]

HOL requires us to give a type to the conditional in the specification of \texttt{FD1}. The type of \( i \) (\( q \) and \( qn \)) is of course \texttt{num->bool} since they each take a time index of type \texttt{num} and map it into a \texttt{bool} value.
FD2 — a 1-bit D-type with asynchronous clear. We now specify a more useful D-type, with asynchronous clear.

This device has two control lines, a data input, and two outputs. Its outputs are undefined at time 0. Thereafter, the output on \( qn \) is always the inverse of that on \( q \). It is simple to specify the outputs on \( q \) and \( qn \) at time \( t+1 \) by cases:

1. F if \( clr \ t \) is high regardless of whether the device is clocked or not at time \( t \)
2. \( \neg t \) if \( clr \ t \) is low and \( clk \ t \) is high
3. \( q \ t \) otherwise (when \( clr \ t \) is low and \( clk \ t \) is low)

a style that maps directly into HOL as

```haskell
#let FD2 = new_definition
('FD2',
 "FD2 clk clr i q qn
  = \t:num.
    let res = ( clr t => F | clk t => \ t | q t )
    in 
      ( (q (t+1)) = res ) /
      ( (qn(t+1)) = ~res )
  );
FD2 = ...
```

We now test this specification to make sure that these conditions hold. Typing requires us to use \texttt{PWR} and \texttt{GND} and not \texttt{T} and \texttt{F} as arguments since both \( clk \) and \( clr \) are of type \texttt{num\rightarrow bool}.  

\[ i \]
\[ \text{FD2 clk clr i q qn} \]
\[ \text{clk} \]
\[ \text{clr} \]
\[ qn \quad q \]
It is manifestly clearer to unfold the `let` at this stage.

For contrast, here is the specification and unfolding of a similar device but with synchronous clear:

```ocaml
#let FD2_1 = new_definition ('FD2_1',
  "FD2_1 clk clr i q qn
  = !t:num.
    let res = ( clk t => (clr t => F | i t) |
          q t
    )
  in
  ( (q (t+1) = res) /\ 
    (qn(t+1) = ~res)
  )
  ");
FD2_1 = ...
```
As corroborated by the rewrites, this specification tells us that the values on \( q \) and \( qn \) will not change unless the device is clocked, and when it is clocked, that the signal on \( clk \) takes precedence over the signal on \( clr \).

### 16.2 Non-primitive circuits

In this section we specify and define implementations for three archetypal clocked sequential circuits — a 1-bit register from which we build a word register and a delay-by-\( n \) circuit. These examples cover most of the basic techniques required when tackling quite advanced sequential circuits.

#### 16.2.1 Definition of Reg

We specify, design and verify a docked, synchronous, 1-bit register which, from dock tick to dock tick, may be cleared to zero, load, or remain as it...
is. The current state of the register is available on output line q; its inverse is available on qn.

The register has an explicit clock line and remains unchanged unless this clock line goes high. If the clock is high at time \( t, t > 0 \), then

- if \( \text{clr} \) is high, the register is cleared to \( F \),
- if \( \text{ld} \) is high, the register is loaded from input line \( i \),
- else the register maintains its previous state.

We can express the logic of this table in HOL in several ways. Here is our chosen specification:

```haskell
#let Reg_spec = new_definition
('Reg_spec'),
"Reg_spec clk clr ld i q qn
= ! t : num .
  let res = ( clk t => ( clr t => F | ld t => i t | q t )
    | q t )
in
  ( q (t+1) = res ) /
  ( qn (t+1) = ~ res )

"");
Reg_spec = ...
```

and here are some tests of this specification which show that state is maintained if \( \text{clk} \) is low, and that \( \text{clr} \) has precedence over \( \text{ld} \).

```haskell
#let RULE
(REWRITE_RULE [ PWR; GND ]
  (SPECL [ "GND"; "(clr:num->bool)"; "(ld:num->bool)" ] Reg_spec));
|! i q qn .
  Reg_spec GND clr ld i q qn =
  (! t. (q(t+1) = q t) \( \land \) (qn(t+1) = ~ q t))

#let RULE
(REWRITE_RULE [ PWR; GND ]
  (SPECL [ "PWR"; "(clr:num->bool)"; "(ld:num->bool)" ] Reg_spec));
|! i q qn .
  Reg_spec PWR clr ld i q qn =
  (! t .
    (q(t+1) = (clr t => F | (ld t => i t | q t)) \( \land \)
    (qn(t+1) = ~ (clr t => F | (ld t => i t | q t))))

Rewriting the definition of \( \text{res} \) to the equivalent form
```
let res = (clk t => p | q t) in
  (q (t+1) = res) /
  (qn(t+1) = ~res)

where p = case (clr t, ld t) of
  (F, F) => q t
  | (F, T) => i t
  | (T, F) => F
  | (T, T) => F

directly suggests an implementation in which clr and ld are control signals to a 4-1 mux whose output leads to a clocked FD1 (see figure 16.5).

```
#let Reg_imp = new_definition
  ('Reg_imp',
   "Reg_imp clk clr ld i q qn
   = ? p . (mux41_imp clr ld i q GND GND p)
   /
   (FD1 clk p q qn)
   ");;
Reg_imp = ...
```

Figure 16.5 1-bit register implementation
16.2.2 Defining nReg

We now specify the behaviour of a clocked word register whose input bits are denoted by \( i_0, i_1, ..., i_n \), and whose output bits are denoted by \( q_0, q_1, ..., q_n \) and \( qn_0, qn_1, ..., qn_n \). As with the 1-bit register, from clock tick to clock tick, the word register may clear to zero (synchronous), parallel load, or remain as is. As we shall see, it may be implemented by a bank of 1-bit registers controlled by the same clock and control lines.

Our problem is one of notation — how may we express the specification clearly and concisely? We adopt the notation \( z_t \ k \) to represent the value on the wire \( z_k \) at time \( t \). Interpret this as \( (z_t) \ k \) — i.e. the \( k \)th wire on bus \( z \) at time \( t \).

To specify the behaviour of the device, we define what is output on lines \( q (t+1) \ k \) and \( qn (t+1) \ k \) (for lines \( k = 0, 1, 2, ..., n \)) at each clock tick \( t \), \( t > 0 \). If we retain the style we used for the 1-bit register, then a first approximation might resemble

```haskell
nReg_spec \ n \ clk \ clr \ ld \ i \ q \ qn
= \ ! k t: num
  let res k = ( clk t => ( clr t => F | ld t => i t k | q t k )
               | q t k )
  in
    ( (q (t+1) k = res) \/
      (qn(t+1) k = ~res) )
```

which holds for all bit lines \( q_k \) and \( qn_k \) with \( k \leq n \). It would be nice if we could raise our level of abstraction and treat the input wires and output wires as though they were bundled together as buses. Noting that all but one of the definitions on the right takes \( k \) as its final argument, we now define the function \( \text{zeros} \ k = F \), and rewrite the first branch on the
right hand side above to \texttt{let res }k = ( \texttt{clk t} =\rightarrow ( \texttt{clr t} =\rightarrow \texttt{zeros }k)). \text{ By the principal of extensionality, we may now “cancel” all indices }k\text{ in the definition of res. }\texttt{nEQ}\text{. will index }k\text{ in the definition of res to the region of interest (}k \leq n\text{). Our specification of this register is now}

\begin{verbatim}
\#let nReg_spec = new_definition
('nReg_spec',
 "nReg_spec n clk clr ld i q qn
 = ! t:num .
 let res = ( clk t =\rightarrow ( clr t =\rightarrow \texttt{zeros } | ld t =\rightarrow i t | q t )
 | q t )
in
 ( (\texttt{nEQL } (q (t+1)) \ res n) \/
 (\texttt{nEQL } (qn(t+1)) \texttt{nNot res } n) )
 ");
nReg_spec = ...
\end{verbatim}

Thanks to our abstractions and extensionality, the specification has the same form as that of the 1-bit register, but the type of the wires has changed from \texttt{num->bool} to \texttt{num->num->bool}.

Our implementation of \texttt{nReg} consists of \((n + 1)\) 1-bit registers wired as shown in figure 16.7 and defined by primitive recursion: a 0-bit register is just a single register.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{nReg.png}
\caption{nReg implementation}
\end{figure}

We build an \((n + 1)\)-bit register by adding one more 1-bit register to an \(n\)-bit system. The definition of this implementation is given as these two parts conjoined together.
As can be seen from the definitions of `Reg_imp` and `nReg_imp`, when we define an implementation, it makes no difference whether we are dealing with combinational or sequential parts. In both cases the arguments are names of wires.

### 16.2.3 Verification of Reg

As our first verification of a sequential circuit we show that our design for `Reg` meets its specification. This proof shows you how to remove hidden lines (simple cases only), cancel $\forall$ quantifications from a conclusion which is a relation, and one way of handling control structures (nested IFs).

Assuming our previous definitions are to hand, we set the goal and unfold.
We now want to eliminate the hidden line \( p \). The sub-components are all combinational and we find that textual occurrences of values on hidden lines on right hand sides match their defining occurrences on left hand sides exactly, time index and all (e.g. \( p t \) in both places). The conversion \texttt{UNWIND\_AUTO\_CONV THEN PRUNE\_CONV} will do this nicely, and it is easy to write a derived rule and a tactic (to traverse theorems and goals respectively).

\begin{verbatim}
#let EXISTS\_ELIM\_RULE
  = CONV\_RULE(DEPTH\_CONV(CHANGED\_CONV(UNWIND\_AUTO\_CONV THENC PRUNE\_CONV)));
EXISTS\_ELIM\_RULE = |- (thm -> thm)

#let EXISTS\_ELIM\_TAC
  = CONV\_TAC(DEPTH\_CONV(CHANGED\_CONV(UNWIND\_AUTO\_CONV THENC PRUNE\_CONV)));
EXISTS\_ELIM\_TAC = |- tactic
\end{verbatim}
We would like to remove the $\forall t$ from both sides of the conclusion of the goal so that $t$ will become free and we can carry out substitutions. The built-in conversion `FORALL_EQ_CONV` takes a term of the generic form $\forall t. \text{lhs} t = \text{rhs} t$ to the theorem

$$|- \ (\forall t. \text{lhs} t = \text{rhs} t) = (\text{lhs} t = \text{rhs} t)$$

`FORALL_EQ_RULE` and `FORALL_EQ_TAC` are also built-in.

At this stage we could unfold the `let`s and bool-case on `clr t` and `ld t`, but we prefer to undertake some algebraic manipulations which turn the `IF` structure on the left (stemming from the implementation) into a semantically equivalent structure matching the pattern on the right.
Since we have carried out this proof by bool-casing, it is fair to ask what we have won. Certainly the main proof will be slightly quicker since the rewriting is restricted, but the main point is that it often pays to simplify the complicated-looking control structures that arise when we rewrite the implementation definitions on the left. If we can rewrite them to the same shape as that given in the specification, we may find terms cancelling without the need for bool-casing. The strategy is used to advantage in our later proof of a counter. When we rewrite with \texttt{lem}

\begin{verbatim}
#let lem = prove
("! t:nnn .
  ( (`clr t /\ `ld t) => q t
   | (`clr t /\ `ld t) => i t
   | ( clr t /\ `ld t) => F
   | F
   ) = (clr t => F | (ld t => i t | q t))",
  GEN_TAC
 THEN MAP_EVERY BOOL_CASES_TAC
 [ (`` (cir:nn->bool) t``; ``(ld:nn->bool) t`` ]
 THEN REWRITE_TAC [ ];

lem =
|! t.
  ((`clr t /\ `ld t) =>
    q t |
    (`clr t /\ `ld t) => i t | (`clr t /\ `ld t) => F | F)) =
  (cir t => F | (ld t => i t | q t))
\end{verbatim}

we find a tautology and there is no more real work to be done.
16.2.4 Verification of nReg

We now verify a typical regular structure, nReg. In this example we unfold let definitions, show the use of the abstractions nEQ and unbundle.

Assuming the availability of the specification and implementation of nReg, we might be tempted to set as goal

```plaintext
# e(REFL_TAC);;
OK..
goal proved

% << **** trace omitted **** >> %

|- !clk clr ld i q qn.
    Reg_imp clk clr ld i q qn = Reg_spec clk clr ld i q qn

Previous subproof:
goal proved
() : void
```

Here is the proof in tidy form.

```plaintext
#let Reg_correct = prove_thm
    ('Reg_correct',
    " ! clk clr ld i q qn .
        Reg_imp clk clr ld i q qn
            = Reg_spec clk clr ld i q qn",
    REPEAT GEN_TAC
    THEN REWRITE_TAC
        [ Reg_imp; Reg_spec;
            mux41_correct; mux41_spec; FD1; GND
        ]
    THEN EXISTS_ELIM_TAC
    THEN FORALL_EQ_TAC
    THEN REWRITE_TAC [ lem ]
    );;
Reg_correct =

|- !clk clr ld i q qn.
    Reg_imp clk clr ld i q qn = Reg_spec clk clr ld i q qn

Run time: 3.0s
Garbage collection time: 0.8s
Intermediate theorems generated: 1005
```
and rewrite from there:

```
#e(INDUCT_TAC THEN_REPEAT GEN_TAC
    THEN PURE_ASM_REWRITE_TAC
    [ nReg_imp; nReg_spec; Reg_correct; Reg_spec; nEQL]);
OK...
2 subgoals
%

"(!t.
    let res = (clk t => (clr t => F | (ld t => i t | ♦ t)) | ♦ t)
    in
    ((q 0(t + 1) = res) \ (qn 0(t + 1) = ¬res)) =

(!t.
    let res = (clk t => (clr t => zeros | (ld t => i t | ♦ t)) | ♦ t)
    in
    ((q(t + 1)0 = res 0) \ (qn(t + 1)0 = ¬not res 0)))"
)

() : void
```

The trouble is that we have a relations for q 0 (t+1) on the left arising from the implementation, and a relation for q (t+1) 0 on the right arising from the specification. Clearly the implementation thinks of q as of type index→time→bool, and the specification thinks of q as of type time→index→bool. Why this confusion? Our natural desire for a high level of abstraction in the specification led us to order the arguments to bus b as in b t k since we can then drop k via extensionality. But in the implementation it is natural to distinguish bus lines by index first b k, and then by time. We can easily make both sides agree in a goal by writing a simple unbundling function

```
#let unbundle = new_definition
  ('unbundle',
   "unbundle (sig:num->num->bool) t n = sig n t");
unbundle = |~ 'sig t n. unbundle sig t n = sig n t
```

and taking the goal as

```
# " ! n clk clr ld i q qn .
  nReg_imp n clk clr ld i q qn
  = nReg_spec n clk clr ld (unbundle i) (unbundle q) (unbundle qn)");

" ! n clk clr ld i q qn .
  nReg_imp n clk clr ld i q qn =
  nReg_spec n clk clr ld(unbundle i)(unbundle q)(unbundle qn)"

() : void
```
Now the proof will go through. We rewrite as before and also unfold the \texttt{lets}.

\begin{verbatim}
% (INDUCT_TAC THEN REPEAT GEN_TAC
  THEN PURE_ASM_REWRITE_TAC
     [ nReg_imp; nReg_spec; Reg_correct; Reg_spec; nEQL ]
  THEN let_TAC);;
OK.
2 subgoals

% << ***** inductive sub-goal omitted **** >> %

"(!t.
  (q 0(t + 1) =
   (clk t => (clr t => F | (ld t => i 0 t | q 0 t)) | q 0 t)) \/
   (qn 0(t + 1) =
    (clk t => (clr t => F | (ld t => i 0 t | q 0 t)) | q 0 t))

(!t.
  (unbundle q(t + 1)0 =
   (clk t =>
    (clr t => zeros | (ld t => unbundle i t | unbundle q t)) |
     unbundle q t)
   O) \/
   (unbundle qn(t + 1)0 =
    nNot
    (clk t =>
     (clr t => zeros | (ld t => unbundle i t | unbundle q t)) |
      unbundle q t)
   O))"

(): void
\end{verbatim}

**Base case.** We attack the base case by cancelling the $\forall \ t$ on both sides.
and then move the argument 0 into the individual arms of the conditional by rewriting the relations for $q$ with \texttt{COND\_RATOR}.

```
#e FORALL_EQ_TAC ;;
OK .
"(q O(t+1) =
  (clk t => (cl t => F | (ld t => i 0 t | q 0 t)) | q 0 t) /
  (qm 0(t+1) =
  "(clk t => (cl t => F | (ld t => i 0 t | q 0 t)) | q 0 t) =
  (unbundle q(t+1)O =
  (clk t =>
  (cl t => zeros | (ld t => unbundle i t | unbundle q t)) |
  unbundle q t)
  0) /
  (unbundle qm(t+1)0 =
  nNot
  (clk t =>
  (cl t => zeros | (ld t => unbundle i t | unbundle q t)) |
  unbundle q t)
  0)")
()
```

All on \texttt{unbundle} now have the requisite number of arguments, and now we may complete our rewriting.

```
#e PURE_REWRITE_TAC [ nNot; COND\_RATOR; zeros; unbundle ];;
OK .
"(q 0(t+1) =
  (clk t => (cl t => F | (ld t => i 0 t | q 0 t)) | q 0 t) /
  (qm 0(t+1) =
  "(clk t => (cl t => F | (ld t => i 0 t | q 0 t)) | q 0 t) =
  (clk t =>
  (cl t => F | (ld t => i 0 t | q 0 t)) | q 0 t)) /
  (qm 0(t+1) =
  "(clk t => (cl t => F | (ld t => i 0 t | q 0 t)) | q 0 t))"
()
```

\textbf{Inductive step.} The inductive subgoal is very long, but inspection shows that the \texttt{nEQL} terms will cancel.

```
#e REFL_TAC ;;
OK .
goal proved
%
```

% << ***** trace emitted ***** >> %
Previous subproof:

```
(\!t.
  nRQL
  (unbundle q(t + 1))
  (clk t =>
   (clr t => zeros | (ld t => unbundle i(t, t) | unbundle q(t)) | unbundle q t))
  n /
)
(nRQL
  (unbundle qn(t + 1))
  (n\not
   (clk t =>
    (clr t => zeros | (ld t => unbundle i(t, t) | unbundle q(t)) | unbundle q t))
  n) /
)

(\!t.
  (q(SUC n)(t + 1) =
   (clk t =>
    (clr t => F | (ld t => i(SUC n)(t, q(SUC n)(t)) | q(SUC n)(t))) /
    (qn(SUC n)(t + 1) =
     ~(clk t =>
      (clr t => F | (ld t => i(SUC n)(t, q(SUC n)(t)) | q(SUC n)(t))) /
      (q(SUC n)(t)))
  )) /

(\!t.
  (nRQL
   (unbundle q(t + 1))
   (clk t =>
    (clr t => zeros | (ld t => unbundle i(t, t) | unbundle q(t)) | unbundle q t))
   n /
   (unbundle qn(t + 1)(SUC n) =
    (clk t =>
     (clr t => zeros | (ld t => unbundle i(t, t) | unbundle q(t)) | unbundle q t)
     (SUC n)) /
   )
  )
)
(nRQL
  (unbundle qn(t + 1))
  (n\not
   (clk t =>
    (clr t => zeros | (ld t => unbundle i(t, t) | unbundle q(t)) | unbundle q t))
   n /
   (unbundle qn(t + 1)(SUC n) =
    (n\not
     (clk t =>
      (clr t => zeros | (ld t => unbundle i(t, t) | unbundle q(t)) | unbundle q t)
      (SUC n))
    ["!clk clr ld i q qn.
     nReg_imp n clk clr ld i q qn =
     nReg_spec n clk clr ld(unbundle i)(unbundle q)(unbundle qn)"]
  ))
)
```

() : void
All we have to do is make the $\forall$ quantifications agree on both sides. It is perhaps simplest to cancel them. There is a built-in conversion `FORALL_AND_CONV` which takes a term like

$$\neg (t \cdot \text{exp1} \land t \cdot \text{exp2})$$

to a theorem

$$\neg (t \cdot \text{exp1} \land t \cdot \text{exp2}) = (t \cdot \text{exp1} \land \text{exp2}).$$

As usual we write a general rule and a general tactic from the conversion.

```
#let FORALL_AND_RULE
  = CONV_RULE (DEPTH_CONV (CHANGED_CONV FORALL_AND_CONV));;
FORALL_AND_RULE = - : (thm -> thm)

#let FORALL_AND_TAC
  = CONV_TAC (DEPTH_CONV (CHANGED_CONV FORALL_AND_CONV));;
FORALL_AND_TAC = - : tactic
```

Using `AND_FORALL_TAC` we can reshape the left hand side of the goal from a conjunction of $\forall$s, to a single $\forall$ whose body is several conjunctions. Then we will have a single $\forall$ on left and right which will be further simplified by a call on `FORALL_EQ_TAC`. Now the major terms in nEQL may be cancelled from both sides.
and we are left with the same goal we had in the base case but with index \( \text{SUC } n \) replacing 0.
Here is the proof in tidy form.

```
#let nReg_correct = prove_thm
  ('nReg_correct',
   "! n clk clr ld i q qn.
    nReg_imp n clk clr ld i q qn = nReg_spec n clk clr ld (unbundle i) (unbundle q) (unbundle qn)",
   INDUCT_TAC THEN REPEAT GEN_TAC
   THEN PURE_ASM_REWRITE_TAC
   [ nReg_imp; nReg_spec; Reg_correct; Reg_spec; nEQL ]
   THEN AND_FORALL_TAC THEN FORALL_EQ_TAC
   THEN let_TAC
   THEN (CANCEL_CONJ_TAC ORELSE ALL_TAC)
   THEN PURE_REWRITE_TAC [ nNot; COND_RATOR; zeros; unbundle ]
   THEN REFL_TAC
  );;

nReg_correct =
  \(! n clk clr ld i q qn.
    nReg_imp n clk clr ld i q qn = nReg_spec n clk clr ld (unbundle i) (unbundle q) (unbundle qn)
  
Run time: 3.2s
Garbage collection time: 0.8s
Intermediate theorems generated: 626
```

### 16.2.5 Line of delays

Our next example verifies a line of FD1s shown in figure 16.8, with the outgoing q line from one device feeding into the input i of the next. The qn lines are not used. We assume a clock that always fires and that the zero-order case is just a straight wire.

Since it has but one input and one output, the specification and implementations of nDel are fairly obvious:
The base case is implemented as just a wire. The other new thing to notice is that we deal with lines that are not used in an implementation, i.e., the corresponding lines lead out but not feed in anywhere. We deal such lines (here \( qn \)) by by existentialising them. When we unfold the implementation, a term of the form \( \exists \ qn . \ \forall \ t . \ qn \ (t+1) = \text{expr} \) will arise and no other term will contain \( qn \). It is trivially easy to satisfy such terms using \texttt{EXISTS_TAC}.

We start the proof by setting the goal and rewriting:

```
16.2. NON-PRIMITIVE CIRCUITS

Base case. Rewriting with ADD_CLAUSES will blow away the base case. We also put the inductive sub-goal in standard SUC form using the "reversed" form of ADD1.

Inductive step. When we unfold the let definition
we are faced with a goal that cannot be handled by \texttt{EXISTS\_ELIM\_TAC} — it cannot cope with terms like \(q(\text{SUC } t) = \ldots\) — and unfortunately, this situation is quite common. A robust strategy is to split the goal into two implications by applying \texttt{EQ\_TAC}, stripping, and then rewriting from the assumptions. Here it is in slow motion.

```lisp
; @e (EQ\_TAC THEN STRIP\_TAC) ;;
; goal proved
; 2 subgoals
"? p qn.
  (!t. p(t+n) = i t) \& (\!t. (q(\text{SUC } t) = p t) \& (qn(\text{SUC } t) = \neg p t))
  [ "!t. nDel\_spec n \& q n Del\_imp n \& !q n Del\_spec n \& !q"
    [ "!t. q(\text{SUC } t + n) = i t'" ]
  [ "!t. q(\text{SUC } t) = i t" ]

  [ "!t. nDel\_imp n \& i q = nDel\_spec n \& !q"
    [ "!t. p(t+n) = i t'" ]
  [ "!t. p(t+n) = i t'
    [ "!t. (q(\text{SUC } t) = p t) \& (qn(\text{SUC } t) = \neg p t)" ]

() : void
```

This goal is solved by rewriting with the assumptions. As we see below, \(\!t. q(\text{SUC } t) = p t\) will rewrite the goal to \(\!t. p(t+n) = i t\) which is echoed on the assumption list.

```lisp
; @e (ONCE\_AS M\_REW RIT E\_TAC}) ;;
; goal proved
; 2 subgoals
"? p qn.
  (!t. p(t+n) = i t) \& (\!t. (q(\text{SUC } t) = p t) \& (qn(\text{SUC } t) = \neg p t))
  [ "!t. nDel\_spec n \& q n Del\_imp n \& !q n Del\_spec n \& !q"
    [ "!t. q(\text{SUC } t + n) = i t'" ]
  [ "!t. q(\text{SUC } t) = i t" ]

  [ "!t. nDel\_imp n \& i q = nDel\_spec n \& !q"
    [ "!t. p(t+n) = i t'" ]
  [ "!t. p(t+n) = i t'
    [ "!t. (q(\text{SUC } t) = p t) \& (qn(\text{SUC } t) = \neg p t)" ]

() : void
```

The second sub-goal is rather more interesting — indeed it is the prime
reason for choosing this example. First note that the unused line `qn` will cause no problems since we can always satisfy `? qn. qn(SUC t) = "p t.`

The harder part is finding a way of matching the signal on `p` coming out of `nDel_imp` (given by `! t . p(t + n) = i t`) with the input to the `(n + 1)/h FD1` (given by `! t. q(SUC t) = p t`). We are going to use `EXISTS_TAC` to turn one of these terms into a tautology, and it makes most sense to focus on the second conjunct (derived from the definition of the simple box `FD1`) rather than looking at substituting using the first conjunct (which is derived from the line of boxes `nDel_imp`). We mentally reverse the relation and treat it as a definition for `p t`. Then we can use `\lambda` to define `p` by `p t = q(SUC t) \equiv p = \lambda x . q (SUC x)`. We now use `EXISTS_TAC` to substitute this definition for `q` into the goal and apply `BETA_TAC` to carry out the \beta-conversion.

```
#e(EXISTS_TAC "\x. (q:num->bool) (SUC x)" THEN BETA_TAC);;
OK.
"?qn.
((\t. q(SUC(t + n)) = i t) \/
 (\t. (q(SUC t) = q(SUC t)) \& (qn(SUC t) = "q(SUC t)"))
 [ "!i q. nDel_imp n i q = nDel_spec n i q" ]
 [ "!t. q(SUC(t + n)) = i t" ]
()
: void
```

Removing `qn` is equally simple and rewriting from the assumptions completes the job.

```
#e(EXISTS_TAC "\x. (q:num->bool) x" THEN BETA_TAC);
OK.
"(!t. q(SUC(t + n)) = i t) \/
 (!t. (q(SUC t) = q(SUC t)) \& (\sim(q(SUC t) = "q(SUC t)"))
 [ "\sim i q. nDel_imp n i q = nDel_spec n i q" ]
 [ "!t. q(SUC(t + n)) = i t" ]
()
: void
```

```
#e(ASM_REWRITE_TAC []);
OK.

% << **** trace omitted **** >> %
|=" i q. nDel_imp n i q = nDel_spec n i q
Previous subproof:
goal proved
()
: void
```
Here is the proof in tidy form.

```plaintext
#let nDel_correct = prove_thm
  "(! n i q . nDel_imp n i q = nDel_spec n i q",
  INDUCT_TAC THEN REPEAT GEN_TAC
  THEN ASM_REWRITE_TAC [ nDel_imp; nDel_spec; PWR ]
  THEN let_TAC
  THEN REWRITE_TAC [ num_CONV "1"; ADD_CLAUSES ]
  THEN EQ_TAC THEN STRIP_TAC
  THEN ASM_REWRITE_TAC []
  THEN EXISTS_TAC "\x. (q:num->bool) (SUC x)" THEN BETA_TAC
  THEN EXISTS_TAC "\x. (q:num->bool) x" THEN BETA_TAC
  THEN ASM_REWRITE_TAC []);

nDel_correct = |- !n i q. nDel_imp n i q = nDel_spec n i q

Run time: 1.7s
Garbage collection time: 0.8s
Intermediate theorems generated: 404
```
EXERCISES 16

Exercise 16.1 Specify the basic parts

\textbf{GND} = \texttt{!t. GND t = F}
\textbf{PWR} = \texttt{!t. PWR t = T}

\texttt{inv = \texttt{!i z. inv i z = (!t. z t = } \sim i t)}
\texttt{and4 = \texttt{!i0 i1 i2 i3 z. and4 i0 i1 i2 i3 z = (!t. z t = i0 t \land i1 t \land i2 t \land i3 t)}}

\texttt{and3 = \texttt{!i0 i1 i2 z. and3 i0 i1 i2 z = (!t. z t = i0 t \land i1 t \land i2 t)}}
\texttt{and2 = \texttt{!i0 i1 z. and2 i0 i1 z = (!t. z t = i0 t \land i1 t)}}

\texttt{nand4 = \texttt{!i0 i1 i2 i3 z. nand4 i0 i1 i2 i3 z = (!t. z t = \lnot (i0 t \land i1 t \land i2 t \land i3 t))}}

\texttt{nand3 = \texttt{!i0 i1 i2 z. nand3 i0 i1 i2 z = (!t. z t = \lnot (i0 t \land i1 t \land i2 t))}}
\texttt{nand2 = \texttt{!i0 i1 z. nand2 i0 i1 z = (!t. z t = \lnot (i0 t \land i1 t))}}

\texttt{nor4 = \texttt{!i0 i1 i2 i3 z. nor4 i0 i1 i2 i3 z = (!t. z t = \lnot (i0 t \lor i1 t \lor i2 t \lor i3 t))}}

\texttt{nor3 = \texttt{!i0 i1 i2 z. nor3 i0 i1 i2 z = (!t. z t = \lnot (i0 t \lor i1 t \lor i2 t))}}
\texttt{nor2 = \texttt{!i0 i1 z. nor2 i0 i1 z = (!t. z t = \lnot (i0 t \lor i1 t))}}

\texttt{or4 = \texttt{!i0 i1 i2 i3 z. or4 i0 i1 i2 i3 z = (!t. z t = i0 t \lor i1 t \lor i2 t \lor i3 t)}}

\texttt{or3 = \texttt{!i0 i1 i2 z. or3 i0 i1 i2 z = (!t. z t = i0 t \lor i1 t \lor i2 t)}}
\texttt{or2 = \texttt{!i0 i1 z. or2 i0 i1 z = (!t. z t = i0 t \lor i1 t)}}

\textbf{FD1}

\texttt{|- !clk i q qn. FD1 clk i q qn = (!t. let res = (clk t => i t \mid q t) in ((q(t+1) = res) \lor (qn(t+1) = \sim res)))}

\textbf{FD2}

\texttt{|- !clk clr i q qn. FD2 clk clr i q qn = (!t.}
Example 16.2.1 Design and verify devices with the following specifications:

```
mux21_spec sel i0 i1 (z:num->bool)
  = ! t:num. z t = (¬sel t) => i0 t | i1 t

mux41_spec sel1 sel2 i0 i1 i2 i3 (z:num->bool)
  = ! t:num.
  z t = (¬sel1 t /\ ¬sel2 t) => i0 t
    | (sel1 t /\ ¬sel2 t) => i1 t
    | (¬sel1 t /\ sel2 t) => i2 t
    |                   i3 t

Reg_spec clk clr ld i q qn
  = ! t:num .
  let res = ( clk t => ( clr t => F
    | ld t => i t
    | q t )
  )
  in
  ( (q (t+1) = res) /\ (qn(t+1) = ¬res) )

nReg_spec n clk clr ld i q qn
  = ! t:num .
  let res = ( clk t => ( clr t => zeros
    | ld t => i t
    | q t )
  )
  in
  ( (nEQL (q (t+1)) res n) /\ (nEQL (qn(t+1)) (NOT res) n) )
```
Chapter 17
Case study I: shifters

17.1 Shifter

In this case study we specify and verify a clocked synchronous shifter that may clear, shift left, shift right, load, or do nothing. We first specify and verify a 1-bit shifter and then the word device. The 1-bit verification is turns out to be trivial; but its specification makes that of the n-bit device trivially easy to get right.

17.1.1 Verification of Shift

This clocked synchronous shifter has separate \( \text{clk} \) and \( \text{clr} \) lines. The signals on the select lines \( \text{sel}_0 \) and \( \text{sel}_1 \) are interpreted according to

<table>
<thead>
<tr>
<th>( \text{sel}_0 )</th>
<th>( \text{sel}_1 )</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>noop</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>shl</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>shr</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>ld</td>
</tr>
</tbody>
</table>

When \( \text{ld} \) is selected, the register reads in its next state on \( i \); when \( \text{shl} \) is selected, the register reads in its next state on \( \text{btm} \); and when \( \text{shr} \) is selected, the register reads in its next state on \( \text{top} \).
Here is our specification of the 1-bit shifter where we have used `let` quite heavily in the interests of readability.

```haskell
#let Shift_spec = new_definition
('Shift_spec',
 "Shift_spec clk clr sel0 sel1 i top btm (q:num->bool) (qn:num->bool)
 = \ t: num .
   let noop = (\(sel0 t) \& \& (sel1 t)) in
   let shl = (\(sel0 t) \& \& (sel1 t)) in
   let shr = (\(sel0 t) \& \& (sel1 t)) in
   let res = (\clk t
                \( c1 r t => F
                \( n o o p => q \ t
                \( s h l => b t m \ t
                \( s h r => t o p \ t
                \( i \ t

            )
            \( q \ t

   )
   in
   (\(q (t+1) = res) \& \& (q (t+1) = 'res)

"");
Shift_spec = ...
```

Rewriting the definition of `res` to the equivalent

```haskell
let res = (clk t => p | q t

in
   (\(q (t+1) = res) \& \& (q (t+1) = 'res)

where p = (\ c l r t => F
       \( case (\sel0 t, \sel1 t) of
       \( F, F => q \ t
       \( F, T => b t m \ t
       \( T, F => t o p \ t
       \( T, T => i \ t

   )
```

directly suggests an implementation in which `clr` is the control signal to a 2-1 mux fed from a 4-1 mux controlled by `sel0` and `sel1` whose output leads to a clocked `FD1`. Here is its definition in HOL:

```haskell
#let Shift_imp = new_definition
('Shift_imp',
 " Shift_imp clk clr sel0 sel1 i top btm (q:num->bool) qn
 = ? p1 p2.
   (mux41_imp sel0 sel1 q btm top i p1) \&
   (mux21_imp clr p1 GWD p2) \&
   (FD1 clk p2 q qn)
"");
Shift_imp = ...
```
The proof is trivial:

```ml
#let lem = prove
("! a (b:*c . ((a) => b | c) = (a => c | b)",
 REPEAT GEN_TAC
 THEN BOOL_CASES_TAC "a:bool"
 THEN REWRITE_TAC []);
lem = |- !a b c . ((`a) => b | c) = (a => c | b)

#let Shift_correct = prove_thm
('Shift_correct',
 " ! clk clr sel0 sel1 i top btm q qn .
 Shift_imp clk clr sel0 sel1 i top btm q qn
 = Shift_spec clk clr sel0 sel1 i top btm q qn",
 REPEAT GEN_TAC
 THEN PURE_REWRITE_TAC
 [ Shift_imp; Shift_spec;
   mux41_correct; mux41_spec;
   mux21_correct; mux21_spec; PDI; GND
 ]
 THEN let_TAC
 THEN EXISTS_ELIM_TAC
 THEN PURE_ONCE_REWRITE_TAC [ lem ]
 THEN REFL_TAC
);
Shift_correct = 
|- !clk clr sel0 sel1 i top btm q qn.
 Shift_imp clk clr sel0 sel1 i top btm q qn =
 Shift_spec clk clr sel0 sel1 i top btm q qn
```

### 17.1.2 Verification of nShift

The specification of nShift is a straightforward extension of the specification of Shift only requiring thought for appropriate definitions for SHL and SHR.

SHL n btm f t takes the bits f t 0, f t 1, ..., f t n and returns f t 1, ..., f t n, btm t.

```ml
#let SHL = new_definition
('SHL',
 "SHL (w:num) new (f:num->num->bool) t k
 = (k = w) => new t | f t (k*i));;

SHL = ...
```

and SHL n top f t takes the bits f t 0, f t 1, ..., f t n and returns top t, f t 0, ..., f t (n-1).
Here is our specification of \textit{nShift}.

\begin{verbatim}
#let nShift_spec = new_definition
('nShift_spec',
  "nShift_spec n clk clr sel0 sel1 i top btm q qn
  = ! t:num .
    let noop = (\(sel0 \ t\) \& \(sel1 \ t\)) in
    let shl = (\(sel0 \ t\) \& \(sel1 \ t\)) in
    let shr = (\(sel0 \ t\) \& \(sel1 \ t\)) in
    let res = ( clk t
        => ( clr t => zeros
             | noop => q t
             | shl => SHL n btm q t
             | shr => SHR n top q t
             | it
             | qt
         )
    )
  in
  ( (nEQL (q (t+1)) res n) \&
    (nEQL (qn(t+1)) (nNot res n))
  )
  ");

nShift_spec = ...
\end{verbatim}

The implementation is simply a row of 1-bit shifters connected internally by the \textit{q} output of device \textit{k} feeding into the \textit{top} input of device \textit{k+1} and into the \textit{btm} input of device \textit{k-1}. Since we want all \textit{q} and \textit{qn} lines to be visible, no existential quantifications are necessary.

\begin{verbatim}
#let nShift_imp = new_prim_rec_definition
('nShift_imp',
  "(nShift_imp 0 clk clr sel0 sel1 i top btm q qn
   = Shift_imp clk clr sel0 sel1 (i 0) top btm (q 0) (qn 0))
  /
  (nShift_imp (SUC n) clk clr sel0 sel1 i top btm q qn
   = nShift_imp n clk clr sel0 sel1 i top (q(SUC n)) q qn
   /
   Shift_imp clk clr sel0 sel1
   (i(SUC n)) (q n) btm (q(SUC n)) (qn(SUC n)))
  )");

nShift_imp = ...
\end{verbatim}

We start the proof by setting the goal
inducting on \( n \) and doing some obvious rewrites. In addition, we clear away the \( \forall s \) and unfold the \( \text{let} \)s.
(unbundle q(t + 1)) o =
(clk t =>
(clr t =>
  zeros |
  (sel 0 \ sel 1) =>
  unbundle q t |
  (sel 0 \ sel 1) =>
  SHR o top(unbundle q t | unbundle i t))) |
  unbundle q t)
0)
(unbundle qn(t + 1)) o =

nNot
(clk t =>
(clr t =>
  zeros |
  (sel 0 \ sel 1) =>
  unbundle q t |
  (sel 0 \ sel 1) =>
  SHR o top(unbundle q t | unbundle i t))) |
  unbundle q t)
0)

() : void

We note that the four major sub-terms in this sub-goal all have the same IF structure. All that remains is to rewrite it into a tautology. We use COND_RATOR to bring the argument 0 into the individual arms of the IFs and unfold a few function definitions. Here is a reminder of the theorems we use:

```ml
#< [ nNot; COND_RATOR; SHL; SHR; unbundle; zeros ];>
[! n a n. nNot a n = ~a n;]
| ! b f g x. (b => f | g)x = (b => f x | g x);]
| ! w new f t k. SHL w new f t k = ((k = w) => new t | f t(k + 1));]
| ! w new f t k. SHR w new f t k = ((k = 0) => new t | f t(k - 1));]
| ! sig t n. unbundle sig t n = sig n t;]
| ! n. zeros n = F]
: the list
```

```ml
#< [ REWRITE_TAC [ nNot; COND_RATOR; SHL; SHR; unbundle; zeros ]];>
OK.
goal proved
```

% << **** trace omitted **** >> %
Previous subproof:

```
"nEQL
(unbundle q(t + 1))
(clk t =>
(clr t =>
zeros |
((sel0 t \ sel1 t) =>
unbundle q t |
((sel0 t \ sel1 t) =>
SIL n(q(SUC n))(unbundle q)t |
((sel0 t \ sel1 t) => SHR n top(unbundle q)t | unbundle i t)))) | unbundle q t)
=\n
nEQL.
(unbundle qn(t + 1))
(n\Not
(clk t =>
(clr t =>
zeros |
((sel0 t \ sel1 t) =>
unbundle q t |
((sel0 t \ sel1 t) =>
SIL n(q(SUC n))(unbundle q)t |
((sel0 t \ sel1 t) => SHR n top(unbundle q)t | unbundle i t)))) | unbundle q t))
=\n
(q(SUC n)(t + 1) =
(clk t =>
(clr t =>
F |
((sel0 t \ sel1 t) =>
q(SUC n)t |
((sel0 t \ sel1 t) =>
btm t |
((sel0 t \ sel1 t) => q n t | i(SUC n)t)))) | q(SUC n)t)) /\
(qn(SUC n))(t + 1) =
~(clk t =>
(clr t =>
F |
((sel0 t \ sel1 t) =>
q(SUC n)t |
((sel0 t \ sel1 t) =>
btm t |
((sel0 t \ sel1 t) => q n t | i(SUC n)t)))) | q(SUC n)t)) =
(nEQL
(unbundle q(t + 1))
(clk t =>
(clr t =>
zeros |
((sel0 t \ sel1 t) =>
```

17.1. **SHIFTER**
The inductive sub-goal is enormous but easy to simplify. First the relations for \( q \) and \( qn \) are just the ones we tackled for the base case with \( \text{SUC } n \) sub-
stituted for 0. After rewriting we cancel them out using \texttt{CANCEL\_CONJ\_TAC}.
In addition to these rewrites we reshape the \texttt{nEQL} terms using \texttt{nEQLThm} and simplify some arithmetic terms arising via \texttt{NOT\_SUC} and \texttt{SUC\_SUB1}.

```plaintext
# [ nEQLThm ; NOT\_SUC ; SUC\_SUB1 ] ;
[ ! \mathit{a} a b . nEQL a b n = (! k k \leq n \implies (a k = b k)) ;
  ! \mathit{a} a . (SUC n = 0) ;
  ! \mathit{m} . (SUC m) - 1 = m ]
: thm list

# (PURE\_ONCE\_REWRITE\_TAC [ nEQLThm ]
THEN REWRITE\_TAC
  [ NOT; COND\_RATOR; SCL; SHR; NOT\_SUC; SUC\_SUB1; zeros; unbundle ]
THEN CANCEL\_CONJ\_TAC) ;
OK.

"(!k.
 k \leq n \implies
 (q k (t + 1)) =
 (clk t =>
  (clr t =>
    F |
    (! sel0 t /\ ~ sel1 t) =>
      q k t |
    (! sel0 t /\ sel1 t) =>
      (k = n) => q(SUC n) t | q(k + 1) t |
    (! sel0 t /\ ~ sel1 t) => (k = 0) => top t | q(k - 1) t | i k t))) |
  q k t )) /

(! k.
 k \leq n \implies
 (qn k (t + 1) =
  (~ clk t =>
    (clr t =>
      F |
      (! sel0 t /\ ~ sel1 t) =>
        q k t |
      (! sel0 t /\ sel1 t) =>
        (k = n) => q(SUC n) t | q(k + 1) t |
      (! sel0 t /\ ~ sel1 t) => (k = 0) => top t | q(k - 1) t | i k t))) |
  q k t ))) /

(! k.
 k \leq n \implies
 (q k (t + 1) =
  (clk t =>
    (clr t =>
      F |
      (! sel0 t /\ ~ sel1 t) =>
        q k t |
      (! sel0 t /\ sel1 t) =>
        (k = SUC n) => btn t | q(k + 1) t |
      (! sel0 t /\ ~ sel1 t) => (k = 0) => top t | q(k - 1) t | i k t))) |
  q k t ))) /
```

17.1. \textit{SHIFTER} 407
The cancelled terms appear on the assumption list — since they are not needed any more, we take the liberty of curtailing their descriptions.

We continue the proof by removing the ∀s and pushing the common antecedent onto the assumption list.
Each sub-term has the same structure and indeed the first 4 cases are already textually identical. A fourfold application of `COND_CASES_TAC THEN AST_REWRITE_TAC []` will do the trick and we write a special tactical to do the job:

```ocaml
#letrec DO n tac = (n = 0) => ALL_TAC | tac THEN (DO (n-1) tac);;
DO = - : (int -> tactic -> tactic)
```
k <= n found on the assumption list is the key to the rest of the proof. It is easy to show that "(k = \text{SUC } n)" and if k < n, then "(k = n). Here are the built-in theorems we need.

If we map \texttt{IMP\_RES\_TAC} over this list, \texttt{LESS\_OR\_EQ} causes a case split, and the two other theorems append just what we want to the assumption list. Here is the essence of the expansion:
17.1. SHIFTER

with inessential assumptions not shown. Rewriting from the assumptions is almost sufficient to solve both goals; we need ⊨ (n = SUC n) and ⊨ SUC n = n + 1 as trivial extras.

Here is the proof in tidy form:
CHAPTER 17. CASE STUDY 1: SHIFTERS

```plaintext
#let nShift_correct = prove_thm
('nShift_correct',
"! n clk clr sel0 sel1 i top btm q qn.
  nShift_imp n clk clr sel0 sel1 i top btm q qn
  = nShift_spec n clk clr sel0 sel1 (unbundle i) top btm (unbundle q) qn)
",
INDUCT_TAC THEN REPEAT GEN_TAC
THEN PURE_ASM_REWRITE_TAC
[ nShift_imp; nShift_spec;
  Shift_correct; Shift_spec;
  nEQL
]
THEN AND_FORALL_TAC THEN FORALL_EQ_TAC
THEN let_TAC
THEN
[ REWRITE_TAC [ nNot; CONDn; SHL; SHR; unbundle; zeros ]

; PURE_ONCE_REWRITE_TAC [ nEQLthm ]
  THEN REWRITE_TAC
[ nNot; CONDn; SHL; SHR; NOT_SUC; SUC_SUB1; zeros; unbundle ]
  THEN CANCEL_CONJ_TAC
  THEN AND_FORALL_TAC THEN FORALL_EQ_TAC
  THEN PURE_REWRITE_TAC[ lem2; lem3 ]
  THEN STRIP_TAC
  THEN (DO 4 (COND_CASES_TAC THEN ASM_REWRITE_TAC [ ]))
  THEN MAP_EVERY IMP_RES_TAC [ LESS_OR_EQ; LESS_SUC; LESS_NOT_EQ ]
  THEN ASM_REWRITE_TAC [ GSYM SUC_ID ; GSYM ADD1 ]
]
)

nShift_correct =
! n clk clr sel0 sel1 i top btm q qn.
  nShift_imp n clk clr sel0 sel1 i top btm q qn
  = nShift_spec
  n clk clr sel0 sel1 (unbundle i) top btm (unbundle q) (unbundle qn)
Run time: 14.9s
Garbage collection time: 10.8s
Intermediate theorems generated: 2531
```
17.2 Staged shifter

17.2.1 Modulo arithmetic

Until now we have deliberately not used modulo arithmetic, deeming it best to get used to "ordinary" arithmetic first. In this example we develop a circular shifter and modulo arithmetic is required. Here are a few of the basic theorems to be found built into HOL.

\[
\begin{align*}
\text{MOD}_{\text{EQ}_0} & = \vdash n. \ 0 < n \implies \ (k \cdot (k \cdot n) \mod n = 0) \\
\text{ZERO}_{\text{MOD}} & = \vdash n. \ 0 < n \implies \ 0 \mod n = 0 \\
\text{MOD}_{\text{PLUS}} & = \vdash n. \ 0 < n \implies \ (\langle \ j \ k \rangle \cdot (\langle \ j \mod n \rangle + (k \mod n))) \mod n \\
& \quad = (j + k) \mod n \\
\text{MOD}_{\text{MOD}} & = \vdash n. \ 0 < n \implies \ (k \cdot (k \mod n) \mod n = k \mod n)
\end{align*}
\]

We prepare for the fact that many of our proofs will be induction proofs by specialising these theorems to the case when \( n > 0 \) by proving some simple variations.

```
#LESS_0;;
\vdash n. \ 0 < (\text{SUC } n)

#let [ \text{MOD}_{\text{EQ}_0}_{\text{SU C}}; \text{ZERO}_{\text{MOD}_{\text{SU C}}}; \text{MOD}_{\text{PLUS}_{\text{SU C}}}; \text{MOD}_{\text{MOD}_{\text{SU C}}} ]
\ = \ \text{map } (\ \text{GEN } "n:\text{num}\")
\quad \text{o } \text{NEWWRITE\_RULE } [ \text{LESS_0 } ]
\quad \text{o } \text{SPEC } "\text{SUC } n"
\)

[ \text{MOD}_{\text{EQ}_0}_{\text{S U C}}; \text{ZERO}_{\text{MOD}}; \text{MOD}_{\text{PLUS}}; \text{MOD}_{\text{MOD}} ];
\text{MOD}_{\text{EQ}_{0_{\text{SU C}}}} = \vdash k. \ (k \cdot (\text{SUC } n)) \ mod (\text{SUC } n) = 0
\text{ZERO}_{\text{MOD}_{\text{SU C}}} = \vdash n. \ 0 \ mod (\text{SUC } n) = 0
\text{MOD}_{\text{PLUS}_{\text{SU C}}} = \\
\quad \vdash n \ j \ k.
\quad \langle j \ mod (\text{SUC } n) \rangle + (k \mod (\text{SUC } n))) \mod (\text{SUC } n) = \langle j + k \rangle \mod (\text{SUC } n)
\text{MOD}_{\text{MOD}_{\text{SU C}}} = \vdash n \ k. \ (k \mod (\text{SUC } n)) \mod (\text{SUC } n) = k \mod (\text{SUC } n)
```

and prove then a useful lemma which is a special case of the second conjunct of the built-in division theorem

```
#DIVISION;;
\vdash n.
\ 0 < n \implies \ (k \cdot (k \cdot \text{DIV } n \cdot n) + (k \mod n)) \land (k \mod n) < n
```
Here is a simple example to get you used to two of our derived lemmata.
We take as goal

\[-! \ a \ b \ m . \ (a + b \ \text{MOD} (\text{SUC} \ m)) \ \text{MOD} (\text{SUC} \ m) = (a + b) \ \text{MOD} (\text{SUC} \ m)\]

remove the quantifications and then rewrite once with the “reverse” of \text{MOD}_\text{PLUS}_\text{SUC}
17.2. STAGED SHIFTER

We now use \texttt{MOD\_MOD\_SUC} to simplify the second of the summed terms on the left

\begin{verbatim}
# MOD\_MOD\_SUC;;
|\! n k. (k MOD (SUC m)) MOD (SUC m) = k MOD (SUC m)

# (PURE\_ONCE\_REWRITE\_TAC [ MOD\_MOD\_SUC ]);;
OK.
"((a MOD (SUC m)) + (b MOD (SUC m))) MOD (SUC m) =
((a MOD (SUC m)) + (b MOD (SUC m))) MOD (SUC m)"
()

() : void
\end{verbatim}

and lo and behold we are left with a tautology

\begin{verbatim}
# e REFL\_TAC;;
OK.
goal proved
|\! (a \+ b) MOD (SUC m) = (a \+ b) MOD (SUC m)

|\! (a MOD (SUC m)) + (b MOD (SUC m))) MOD (SUC m) =
((a MOD (SUC m)) + (b MOD (SUC m))) MOD (SUC m)

|\! (a + (b MOD (SUC m))) MOD (SUC m) = (a + b) MOD (SUC m)
|\! !a b m. (a + (b MOD (SUC m))) MOD (SUC m) = (a + b) MOD (SUC m)

Previous subproof:
goal proved
() : void
\end{verbatim}

Here is the tidied theorem. We will use it twice in deriving the main result of this chapter.

\begin{verbatim}
#define lem = prove
("! a b m. (a \+ b) MOD (SUC m) = (a \+ b) MOD (SUC m)",
REPEAT GEN\_TAC
THEN PURE\_ONCE\_REWRITE\_TAC [ GSYM MOD\_PLUS\_SUC ]
THEN PURE\_ONCE\_REWRITE\_TAC [ MOD\_MOD\_SUC ]
THEN REFL\_TAC);

lem = |\! !a b m. (a + (b MOD (SUC m))) MOD (SUC m) = (a + b) MOD (SUC m)
\end{verbatim}
17.2.2 nMux21

To be removed and made into an exercise.
17.2. STAGED SHIFTER

Implementation

We are going to build a combinational device which shifts its input \( i \) circularly to the right \( \text{val\ shift\ } m \) and emits the result on \( z \). Word \( i \) is of size \( n \) and word \( \text{shift} \) is of size \( m \).

![Figure 17.2 n \times m staged shifter](image)

A semi-formal specification of the device is

\[
\text{SS spec } n \times m \text{ shift } i z = ! t. \ (n \text{EQL} \ (z t) \ \text{rotation} \ n)
\]

where \( \text{rotation} = \text{CIRC} \ (\text{val} \ (\text{shift} t) \ m) \ i \ t \)

We build the device by wiring together a number of \( \text{nbs} \) devices as shown in figure 17.3 where \( \text{nbs} \) box \( k \) either passes its input straight through or else rotates it circularly to the right by \( 2^k \) according to its one bit input \( \text{yes} \). More formally

\[
\text{nbs spec } n \text{ level } \text{yes} \ i z = ! t. \ (n \text{EQL} \ (z t) \ \text{rotation} \ n)
\]

where \( \text{rotation} = (\text{yes} t) \Rightarrow i \ t | \text{CIRC} \ (2 \ \text{EXP} \ \text{level}) \ i \ t \)

Building an \( \text{nbs} \) box is easy — it is just a specialisation of \text{Mux21}.

17.2.3 Verification of \( \text{nbs} \)

```plaintext
#let wCIRC = new_definition
  ("CIRC",
   "wCIRC n m (a:num->num->bool) k t = a ((m*k) MOD (SUC n)) t");

#let nCIRC = new_definition
  ("CIRC",
   "nCIRC n m (a:num->num->bool) t k = a t ((m*k) MOD (SUC n))");
```

Figure 17.3 \( n \times m \) staged shifter implementation

```plaintext
\textbf{#let nbs\_spec = new\_definition}
\hspace{1em} ('nbs\_spec',
\hspace{1em} "nbs\_spec n level yes i z
\hspace{1em} = ! (t: num).
\hspace{1em} (nEQL (z t)
\hspace{1em} ('yes t) => i t | nCIRC n (2 EXP level) i t)
\hspace{1em} n
\hspace{1em} ');
\hspace{1em} nbs\_spec = ...
\textbf{#let nbs\_imp = new\_definition}
\hspace{1em} ('nbs\_imp',
\hspace{1em} "nbs\_imp n level yes i z
\hspace{1em} = nMux21\_imp n yes i (wCIRC n (2 EXP level) i) z");;
\hspace{1em} nbs\_imp = ...
\textbf{#let nbs\_correct = prove\_thm}
\hspace{1em} ('nbs\_correct',
\hspace{1em} "! n level yes i z .
\hspace{1em} nbs\_imp n level yes i z
\hspace{1em} = nbs\_spec n level yes (unbundle i) (unbundle z)",
\hspace{1em} REPEAT GEN\_TAC
\hspace{1em} THEN PURE\_REWRITE\_TAC
\hspace{1em} [ nbs\_imp; nbs\_spec; nMux21\_correct; nMux21\_spec; nEQL\_Thm ]
\hspace{1em} THEN REPEAT FOR\_ALL\_EQ\_TAC
\hspace{1em} THEN REMOTE\_TAC [ NUM\_RATOR; nCIRC; wCIRC; unbundle ]
\hspace{1em});;;
\hspace{1em} nbs\_correct =
\hspace{1em} |- ! n level yes i z .
\hspace{1em} nbs\_imp n level yes i z =
\hspace{1em} nbs\_spec n level yes (unbundle i) (unbundle z)
```

Chapter 17. Case Study I: Shifters
17.2. STAGED SHIFTER

17.2.4 Verification of nSS

```haskell
#let nSS_spec = new_definition
  ('nSS_spec',
   "nSS_spec n m shift i z
     = ! (t:num).
       (nEQL (z t) (nCIRC n (val (shift t) m) i t) n)
     ");
  nSS_spec = ...

#let nSS_imp = new_prim_rec_definition
  ('nSS_imp',
   "(nSS_imp n 0 shift i z
     = nbs_imp n 0 (shift 0) i z)
     /
     (nSS_imp n (SUC m) shift i z
      = ? q . nSS_imp n m shift i q /
        nbs_imp n (SUC m) (shift(SUC m)) q z)
     ");
  nSS_imp = ...
```

We start the verification by inducting on \( m \) and rewriting the top level terms.

```haskell
#g "! n m shift i z.
  nSS_imp n m shift i z
   = nSS_spec n m (unbundle shift) (unbundle i) (unbundle z)";;
"! n m shift i z.
  nSS_imp n m shift i z =
  nSS_spec n m(unbundle shift)(unbundle i)(unbundle z)"

() : void
#e(GEN_TAC THEN IDUCT_TAC THEN REPEAT GEN_TAC
  THEN ASM_REWRITE_TAC
  [ nSS_imp; nSS_spec; nbs_correct; nbs_spec; val ]);;
OK...
2 subgoals

% << **** inductive goal omitted *** >> %

"(! t.
  nEQL
    (unbundle z t)
    (('shift 0 t) => unbundle i t | nCIRC n(2 EXP 0)(unbundle i)t)
  n) =
  (!t. nEQL(unbundle z t)(nCIRC n(bw(unbundle shift t 0))(unbundle i)t)n)"

() : void
Base case.

Since we are not inducting on $n$ we will not be able to rewrite any terms in $\text{nEQL}$ — to make headway we use $\text{nEQLThm}$ and $\text{COND\_RATOR}$ to distribute the argument $n$ inwards, $\text{REPEAT\_FORALL\_EQ\_TAC}$ to remove the universal quantifications arising, $\text{lem2}$ to take the implication outwards, and $\text{STRIP\_TAC}$ to push it onto the assumption list.

We now bool-case on $\text{shift\_0\_t}$. The case $\text{shift\_0\_t} = \text{T}$ is trivial (just rewrite with $\text{bvals}$). When $\text{shift\_0\_t} = \text{F}$, we have to show that $(z\ k\ t = i\ k\ t) = (z\ k\ t = \text{nCIRC\ n\ 0\ (unbundle\ i)\ t\ k})$. Rewriting with $\text{nCIRC}$ expands the rightmost sub-term to $(\text{unbundle\ i})\ t\ ((0+k)\ \text{MOD\ (SUC\ n)})$.
17.2. STAGED SHIFTER

Here are the final steps.

Inductive case.
We start off by rewriting terms in \texttt{nEQ\L}. Automatically removing the hidden line \texttt{q} is beyond the capabilities of \texttt{EXISTS\_ELIM\_TAC} so we resort to \texttt{EQ\_TAC}. 
Here we use the textually lowest two assumptions to derive a relation for \( z \) for substitution into the goal. If we probe via \texttt{RES_TAC}, 6 assumptions are added to the assumption list, amongst them

\[
\begin{align*}
\textbf{then } & \text{_repeat strip_tac};;
\end{align*}
\]

"?q.
\begin{align*}
!t. & \text{ k} \not\leq n \implies (q \ k \ t = i((\text{val}(\text{unbundle shift } t)m) + k) \text{ MOD } (\text{SUC } n) t)) /\\
!t. & \text{ k} \leq n \implies \text{ (z k t} = \text{ (shift(SUC m)t) }\implies q \ k \ t \ | q(((2 \text{ EXP } (\text{SUC m}) + k) \text{ MOD } (\text{SUC } n)t)) ;
\end{align*}
"!shift i z.
\begin{align*}
\text{ nSS_imp n m shift i z} = \\
\text{ nSS_spec n (unbundle shift)(unbundle i)(unbundle z)" ]}
\end{align*}
"!t k.
\begin{align*}
& \text{ k} \not\leq n \implies \text{ (z k t} = 1 \\
& \text{ (((val(unbundle shift t)m) + ((2 EXP (SUC m)) + (bv(shift(SUC m)t)))) + k) MOD (SUC n)) t)"
\end{align*}
"!shift i z.
\begin{align*}
\text{ nSS_imp n m shift i z} = \\
\text{ nSS_spec n (unbundle shift)(unbundle i)(unbundle z)" ]}
\end{align*}
"!t k.
\begin{align*}
& \text{ k} \leq n \implies \text{ (z k t} = i((\text{val}(\text{unbundle shift } t)m) + k) \text{ MOD } (\text{SUC } n) t)"
\end{align*}
"!t k.
\begin{align*}
& \text{ k} \leq n \implies \text{ (z k t} = \text{ (shift(SUC m)t) }\implies q \ k \ t \ | q(((2 \text{ EXP } (\text{SUC m}) + k) \text{ MOD } (\text{SUC } n)t))" ]
\end{align*}
"k \leq n"
(0) : void

Here we use the textually lowest two assumptions to derive a relation for \( z \) for substitution into the goal. If we probe via \texttt{RES_TAC}, 6 assumptions are added to the assumption list, amongst them

\[
\begin{align*}
\textbf{then } & \text{_repeat strip_tac};;
\end{align*}
\]

"?q.
\begin{align*}
!t. & \text{ k} \not\leq n \implies (q \ k \ t = i((\text{val}(\text{unbundle shift } t)m) + k) \text{ MOD } (\text{SUC } n) t)) /\\
!t. & \text{ k} \leq n \implies \text{ (z k t} = \text{ (shift(SUC m)t) }\implies q \ k \ t \ | q(((2 \text{ EXP } (\text{SUC m}) + k) \text{ MOD } (\text{SUC } n)t)) ;
\end{align*}
"!shift i z.
\begin{align*}
\text{ nSS_imp n m shift i z} = \\
\text{ nSS_spec n (unbundle shift)(unbundle i)(unbundle z)" ]}
\end{align*}
"!t k.
\begin{align*}
& \text{ k} \not\leq n \implies \text{ (z k t} = 1 \\
& \text{ (((val(unbundle shift t)m) + ((2 EXP (SUC m)) + (bv(shift(SUC m)t)))) + k) MOD (SUC n)) t)"
\end{align*}
"!shift i z.
\begin{align*}
\text{ nSS_imp n m shift i z} = \\
\text{ nSS_spec n (unbundle shift)(unbundle i)(unbundle z)" ]}
\end{align*}
"!t k.
\begin{align*}
& \text{ k} \leq n \implies \text{ (z k t} = \text{ (shift(SUC m)t) }\implies q \ k \ t \ | q(((2 \text{ EXP } (\text{SUC m}) + k) \text{ MOD } (\text{SUC } n)t))" ]
\end{align*}
"k \leq n"
(0) : void

Here we use the textually lowest two assumptions to derive a relation for \( z \) for substitution into the goal. If we probe via \texttt{RES_TAC}, 6 assumptions are added to the assumption list, amongst them

\[
\begin{align*}
\textbf{then } & \text{_repeat strip_tac};;
\end{align*}
\]
We use the theorem continuation `RES_THEN` to generate the 6 theorems and then rewrite with them (only one can take effect).

```plaintext
#e(RS_THEN (\th. PURE_ONCE_REWRITE_TAC [ th ]));
OK...
"((\text{shift}(\text{SUC} \ m) t) \Rightarrow
 i(((\text{val}(\text{unbundle shift} \ m) + k) \text{ MOD} (\text{SUC} \ n)) t) \Rightarrow
 q(((2 \text{ EXP} (\text{SUC} \ m)) + k) \text{ MOD} (\text{SUC} \ n)) t) =
 i
(((\text{val}(\text{unbundle shift} \ m)) + ((2 \text{ EXP} (\text{SUC} \ m)) \ast (\text{bv(shift}(\text{SUC} \ m) t)))) + k) \text{ MOD}
(\text{SUC} \ n))
 t"[
 "!\text{shift} i z.
 nSS_imp n m shift i z =
 nSS_spec n m(\text{unbundle shift})(\text{unbundle i})(\text{unbundle z})"
 ]
[ "!t k.
 k <\text{=} n \Rightarrow
 (q k t = i(((\text{val}(\text{unbundle shift} \ m) + k) \text{ MOD} (\text{SUC} \ n)) t))"
 ]
[ "!t k.
 k <\text{=} n \Rightarrow
 (z k t =
 ((\text{shift}(\text{SUC} \ m) t) \Rightarrow
 q k t)
 q(((2 \text{ EXP} (\text{SUC} \ m)) + k) \text{ MOD} (\text{SUC} \ n)) t))"
 ]
[ "k <\text{=} n"
 ]
()
 : void
```

The obvious method of attack here is by cases on `\text{shift}(\text{SUC} \ m) t`. The case `\text{shift}(\text{SUC} \ m) t = F` is trivial.
The case \( \text{shift} \ (\text{SUC} \ m) \ t = T \) leaves us with \( q \ k \ t \) in the goal and to get rid of that we will need to use the assumption

\[
q(k = n) = ((\text{val}(\text{unbundle shift} t) m) + k) \mod (\text{SUC} n) t
\]

\[
q(k = n) = ((\text{val}(\text{unbundle shift} t) m) + k) \mod (\text{SUC} n) t
\]
All that now remains is a simple rewrite with `lem` to simplify the left hand side and to put the summed terms in normal order.

We start the last leg of the proof by trivialising the relation for `q k t`. 
and then we put this sub-goal into the same shape as the previous one expecting that the same style of attack will prevail.
Again we use the textually bottommost two assumptions to rewrite \( z \ k \ t \). To make sure we probe the goal with RES_TAC first. Three new assumptions are generated, amongst them

\[
!t.\ z \ k \ t = \\
(((\text{val}(\text{unbundle shift } t)m) + k) \mod (\text{SUC } n)) t \\
\]

with which we will rewrite. Using RES_THEN enables us to do the rewrite without cluttering the assumption list.
When we bool-case on \texttt{shift (SUC m)} \texttt{t}, one case is trivial
and the other one we have seen before.

```plaintext
#e(BOOL_CASES_TAC "(shift:num->num->bool) (SUC m) t"
   THEN REWRITE_TAC [ bvals; MULT_CLAUSES; ADD_CLAUSES ]);
OK

"i(((val(unbundle shift t)m) + (2 ** (SUC m))) + k) MOD (SUC n))t =
 i (((val(unbundle shift t)m) + (((2 ** (SUC m)) + k) MOD (SUC n))) MOD (SUC n))
 t"
 [ "!shift i z.
   nSS_imp n m shift i z =
   nSS_spec n m(unbundle shift)(unbundle i)(unbundle z)"
 ]
 [ "t k.
   k <= n ==> 
   (i k t =
    i (((val(unbundle shift t)m) +
     ((2 ** (SUC m)) * (bv(shift(SUC m)t)))) +
     k) MOD
     (SUC n))
     t")"
 ]
 [ "k <= n" ]
)
 : void
```

```plaintext
#e(REWRITE_TAC [ lem; ADD_ASSOC ]);
OK.
goal proved

% << ***** trace omitted ***** >> %

\[= !n m shift i z.
   nSS_imp n m shift i z =
   nSS_spec n m(unbundle shift)(unbundle i)(unbundle z)\]

Previous subproof:
goal proved
() : void
```
Here is the proof in tidy form.

```plaintext
#let nSS_correct = prove_thm
  ('nSS_correct',
  "! n m shift i z.
   nSS_imp n m shift i z
   = nSS_spec n m (unbundle shift) (unbundle i) (unbundle z)",
  GEN_TAC THEN INDUCT_TAC THEN REPEAT GEN_TAC
  THEN ASM_REWRITE_TAC
  [ nSS_imp; nSS_spec; nSS_correct; nss_spec; val ]
  THENL
  [ % base case %
    PURE_REWRITE_TAC [ nEQLThm; COND_RATOR; EXP; unbundle ]
    THEN REPEAT FORALL_EQ_TAC
    THEN PURE_ONCE_REWRITE_TAC [ lem2 ]
    THEN STRIP_TAC
    THEN MAP_EVERY IMP_REWRITE_TAC
    THEN PURE_ONCE_REWRITE_TAC [
        nEQLThm; COND_RATOR; unbundle; nCIRC
    ]
    THEN EQ_TAC
    THEN REPEAT STRIP_TAC
    THENL
      [ RES_THEN (\th. PURE_ONCE_REWRITE_TAC [ th ])
        THEN BGGC_CASES_TAC "(shift:num->num->bool) 0 t"
        THEN REWRITE_TAC [ bvals; MULT_CLAUSES; ADD_CLAUSES ]
      ;
      % inductive case %
        RES_THEN (\th. PURE_ONCE_REWRITE_TAC [ th ])
        THEN BGGC_CASES_TAC "((2 EXP (SUC m)) * k)";
        "n: num" [ MOD_BOUND ]
        THEN REWRITE_TAC [ lem; ADD_ASSOC ]
      ]
  ];

nSS_correct =
  |- ! n m shift i z.
   nSS_imp n m shift i z
   = nSS_spec n m (unbundle shift) (unbundle i) (unbundle z)

Run time: 5.7s
Garbage collection time: 3.8s
Intermediate theorems generated: 1343
```
Chapter 18

Case studies II: counter

18.1 Case II: up counter

We specify and verify an asynchronous up counter taken from [36, page 401].

18.1.1 1-bit up counter

Here is a black box view of the 1-bit device.

\[
\begin{array}{cccc}
\text{clk} & \text{clr} & \text{ld} & \text{i} \\
\downarrow & \downarrow & \downarrow & \downarrow \\
\text{ce} & \text{UpCount} & \text{clk} & \text{clr} & \text{ld} & \text{i} & \text{q} & \text{qn} & \text{tc} \\
\downarrow & \downarrow & \text{q} \\
\end{array}
\]

Figure 18.1 1-bit up counter

At any time \( t \), the device may be cleared if \( \text{clr} t \) is high. Otherwise, it may be loaded with a fresh value on \( i \) if \( \text{ld} t \) is high, incremented if \( \text{ce} t \) is high, or remain as is. Our first informal specification for the signal on \( q \) reads

\[
q(t + 1) = \begin{cases} 
\text{clr} t & \Rightarrow P \\
\text{clk} t & \Rightarrow ( \text{ld} t \Rightarrow i t \\
\text{ce} t & \Rightarrow \overline{q} t \\
q t & \end{cases}
\]

in which the clear operation takes place even if \( \text{clk} t \) is not high since the device is asynchronous, and the load operation takes precedence over the count operation. Since it is convenient if the definition of the bit device
and the word device have the same form, we choose to look at the value on q through num eyes and write

\[
\begin{align*}
\text{bv}(q(t+1)) &= \text{clr} t \Rightarrow \text{bv} 0 \\
& \quad | \text{clk} t \Rightarrow (\text{ld} t \Rightarrow \text{bv}(i t) \\
& \quad \quad | \text{ce} t \Rightarrow (\text{bv}(q t) + 1) \mod 2 \\
& \quad \quad \quad | \text{bv}(q t) \\
& \quad | \text{bv}(q t)
\end{align*}
\]

Although it does look more cluttered than our previous attempt, this arithmetic formulation does generalise nicely whereas the boolean formulation does not.

\(tc\) is used when several 1-bit devices are chained together to form a word device (it will ripple from the least significant end). It feeds into the \(ce\) of the next device and thus should be high only when the state of this device switches from high to low. It is implemented by \textbf{and} \(ing\) together the \(ce\) and the \(q\) lines. The output from this line at time \(t\) will be used by the next device at time \(t\) to set its state at time \(t+1\), even at time 0. Thus the specification of the bit value on this line is

\[tc \ t = ce \ t \land q \ t\]

Here is our final choice of specification:

```plaintext
#let UpCount_spec = new_definition
('UpCount_spec',
  "UpCount_spec clk clr ce ld i q tc
   = !t .
     ( let VAL = bv(q t) in
       let MAX = 1 in
       ( ( bv(q(t+1))
         = ( clr t => 0
           | clk t
             => ( ld t => bv (i t)
               \ | ce t => ((VAL = MAX) => 0 | VAL+1)
               \ | VAL
             )
           )
         )
       \ /
       (tc t = ce t \ (VAL = MAX))
     )
   ;)
  ");
UpCount_spec = ...
```
where $\text{VAL} = \text{MAX}$ is equivalent to $q_t = T$ or $bV(q_t) = 1$.

The implementation of this device is suggested by our expressing the innards of the relation for $q$ as

$$q(t + 1) = \text{clr} t \Rightarrow F | p$$

where $p = \text{clk} t \Rightarrow \text{res} | q_t$

where $\text{res} = \text{case} (ld_t, ce_t) \text{ of}$

$$\begin{align*}
(F, F) &= q_t &\% \text{no operation} \\
(F, T) &= ~q_t &\% \text{load low but ce high} \\
(T, F) &= i_t &\% \text{load high} \\
(T, T) &= i_t &\% \text{load high}
\end{align*}$$

We implement the case statement by a 4:1 mux which feeds into an FD2 which has an asynchronous clear and remains in the same state when not clocked. We locally invert $q$ to supply $\sim q_t$. The relation for $tc$ is just an \text{and} gate.

```plaintext
#let UpCount_imp = new_definition
('UpCount_imp',
  "UpCount_imp clk clr ce ld i q tc
  = ? qn p qBAR .
  (inv q qBAR)  \/
  (mux41_imp ld ce q qBAR i i p) \/
  (FD2 clk clr p qqn)  \/
  (and2 ce q tc)
  ");;
UpCount_imp = ...
```

Implementing the case statement more efficiently using \text{and} and \text{or} gates is left as an exercise.

**Verification**

This turns out to be the most interesting verification of a 1-bit device in this book! We set the goal and do the expected rewrites.

```plaintext
#g " ! clk clr ce ld i z tc .
UpCount_imp clk clr ce ld i z tc
  = UpCount_spec clk clr ce ld i z tc";;
"! clk clr ce ld i z tc .
UpCount_imp clk clr ce ld i z tc = UpCount_spec clk clr ce ld i z tc";;

() : void
```

```plaintext
#e(REPEAT GEN_TAC
  THEN PURE_REWRITE_TAC
  [ UpCount_imp; UpCount_spec; FD2;
    mux41_correct; mux41_spec; inv; and2 ]);
```
Two of the hidden lines may removed by applying `EXISTS_ELIM_TAC` then we unfold the `lets`.
We now aim to cancel the relations for tc.\textsuperscript{1} We bring the $\forall$ in on the right hand side using \texttt{FORALL\_AND\_TAC}, and rewrite $bv(z(t)) = 1$ to $zt$ using \texttt{ivals}.

\begin{verbatim}
#e(FORALL_AND_TAC THEN REWRITE_TAC [ ivals ]);;
OK. \\
% \(\text{**** lhs of goal omitted \text{****}}\) >> %

(!t.
  bv(z(t + 1)) =
  (clr t =>
   0 |
   (clk t =>
    (ld t => $bv(i t) |
     (ce t => $bv(z t) => 0 | ($bv(z t) + 1) | $bv(z t) ) ) |
    $bv(z t ))) /\ 
  (!t. tc t = ce t /\ z t ) )

() : void
\end{verbatim}

We now have a relation of the form $\exists qn . A(qn) \land B$ on the left where $A$ is a function of $qn$ and $B$ is not. The conversion \texttt{EXISTS\_AND\_CONV} takes such a term and returns the theorem.

\textsuperscript{1}In this example, this turns out very much to be art for art's sake as the relation in question cancels itself would cancel itself out later when we apply \texttt{EQ_TAC THEN \_STRIP_TAC THEN ASM\_REWRITE\_TAC []}. However we persevere in the belief that this trick is generally useful.
\[-(? \text{qn} . \ A(\text{qn}) \lor B) = (? \text{qn} . \ A(\text{qn})) \lor B\]

(Aside: \textsc{exists\_and\_conv} also copes with terms like \(? \text{qn} . \ A \lor B(\text{qn}).\)

We apply \textsc{exists\_and\_conv} and cancel the relations for \(tc\).

```
#e(CONV_TAC(CHANGED_CONV(ONCE_DEPTH_CONV EXISTS_AND_CONV)))
THEN CANCEL_CONV_TAC;;
OK.
"(?\text{qn}.
  (!t.
    \text{z}(t + 1) =
    (\text{clr} \ t \Rightarrow
     F |
     \text{clk} \ t \Rightarrow
     (((\text{ld} \ t \lor \text{ce} \ t) \Rightarrow \text{z} \ t |
       ((\text{ld} \ t \lor \text{ce} \ t) \Rightarrow \text{z} \ t |
        ((\text{ld} \ t \lor \text{ce} \ t) \Rightarrow \text{i} \ t | \text{i} \ t)) |
        \text{z} \ t))) \lor
    (!t.
     \text{qn}(t + 1) =
     \neg(\text{clr} \ t \Rightarrow
     F |
     \text{clk} \ t \Rightarrow
     (((\text{ld} \ t \lor \text{ce} \ t) \Rightarrow \text{z} \ t |
       ((\text{ld} \ t \lor \text{ce} \ t) \Rightarrow \text{z} \ t |
        ((\text{ld} \ t \lor \text{ce} \ t) \Rightarrow \text{i} \ t | \text{i} \ t)) |
        \text{z} \ t))) =
    (!t.
     \text{bv}(\text{z}(t + 1)) =
     (\text{clr} \ t \Rightarrow
     0 |
     \text{clk} \ t \Rightarrow
     (\text{ld} \ t \Rightarrow \text{bv}(\text{i} \ t) | (\text{ce} \ t \Rightarrow (\text{z} \ t \Rightarrow 0 | (\text{bv}(\text{z} \ t)) + 1 | \text{bv}(\text{z} \ t))) | \text{bv}(\text{z} \ t))))"
[ "!t. tc t = ce t \lor z t" ]
() : void
```

Our next step is to make the case structures everywhere the same. All we need are two simple lemmata with which we rewrite purely and once.
We now do some more algebraic manipulations by rewriting the 0 on the right hand side to $\text{bv F}$. Then every arm in the conditional on the right is an application of $\text{bv}$ and we may bring $\text{bv}$ out using $\text{fCOND}$ “in reverse”.
Now the bvs on the right may be cancelled thanks to bvEq1 and we are left with a goal of the form ?qn . A \land B = B. The easiest way of proceeding is to apply EQ_TAC, strip, and rewrite from the assumptions. The terms in B will vanish in both cases and we are left with a straightforward goal.

It is not hard to devise the “right” substitution make this a tautology; rewriting with add_sub simplifies the terms (t + 1) - 1 that arise.
#ADD_SUB;
\[ \vdash !a \ c \ (a \ + \ c) - c = a \]

#e(EXISTS_TAC "\ t . \ ( \ clr \ (t-1) \Rightarrow F \n | \ ( \ clk \ (t-1) \Rightarrow \\
 | \ ( \ ld \ (t-1) \Rightarrow \ i \ (t-1)) \\
 | \ ( \ ce \ (t-1) \Rightarrow \ z \ (t-1) \ | \ z \ (t-1)) \n | \ z \ (t-1) \n) \\
)"

THEN BETA_TAC
THEN REWRITE_TAC [ADD_SUB];;

OK..
goal proved

% << **** trace omitted **** >> %
\[ \vdash !clk \ clr \ ce \ ld \ i \ z \ tc. \n\text{UpCount_imp clk clr ce ld i z tc = UpCount_spec clk clr ce ld i z tc} \]

Previous subproof:
goal proved
() : void
Here is the proof in tidy form:

```ml
#let UpCount_correct = prove_thm
('UpCount_correct',
  "! clk clr ce ld i z tc.
   UpCount_imp clk clr ce ld i z tc
   = UpCount_spec clk clr ce ld i z tc",
REPEAT GEN_TAC
THEN PURE_REWRITE_TAC
  [ UpCount_imp; UpCount_spec; FD2;
    mux41_correct; mux41_spec; inv; and2
  ]
THEN let_TAC
THEN FORALL_AND_TAC
THEN EXISTS_ELIM_TAC
THEN REWRITE_TAC [ ivals ]
THEN MAP_EVERY (\th . PURE_ONCE_REWRITE_TAC [ th ])
  [ lem2; lem3; GSYM (CONJUNCT2 bvals) ]
THEN PURE_REWRITE_TAC [ GSYM (ISPEC "bv" CMD_RAND) ]
THEN PURE_ONCE_REWRITE_TAC [ bvEq1 ]
THEN EQ_TAC THEN REPEAT STRIP_TAC
THEN ASM_REWRITE_TAC []
THEN EXISTS_TAC
  "\t . ~( clr (t-1) => P
    | (clk (t-1) =>
      ( ld (t-1) => i (t-1)
      | (ce (t-1) => "z (t-1) | z (t-1))
    | z (t-1))"
 THEN BETA_TAC
 THEN REWRITE_TAC [ ADD_SUB ];
UpCount_correct = ! clk clr ce ld i z tc.
UpCount_imp clk clr ce ld i z tc = UpCount_spec clk clr ce ld i z tc
Run time: 8.6s
Garbage collection time: 3.3s
Intermediate theorems generated: 2585
```

18.1.2 An implementation that does NOT work

Since \( qn \ t \) is the inverse of \( q \), why did we invert \( q \) in our implementation of \( \text{UpCount} \) and not just feed \( qn \) into the \( \text{mux41} \)? If you cannot guess then try a proof with this implementation:
We set the goal as before and after exactly the same steps arrive at:

```plaintext
#g " ! clk clr ce ld i q tc .
  UpCount_imp' clk clr ce ld i q tc = UpCount_spec clk clr ce ld i q tc";
"!clk clr ce ld i q tc.
  UpCount_imp' clk clr ce ld i q tc = UpCount_spec clk clr ce ld i q tc"
()
```

2 subgoals

```
"?qn.
  (t.
   (c1r t => F | (clk t => (ld t => i t | (ce t => ^q t | q t)) | q t)) =
   (c1r t => F | (clk t => (ld t => i t | (ce t => qn t | q t)) | q t))) \n
  (t.
   qn(t + i) =
   "(c1r t =>
     F | (clk t => (ld t => i t | (ce t => qn t | q t)) | q t)))"
  [ "!t.
   q(t + i) =
   (c1r t =>
     F | (clk t => (ld t => i t | (ce t => ^q t | q t)) | q t))"
  [ "!t. t t = ce t \ q t' ]
```
which are unprovable since we cannot show that $q_0 = \neg qn_0$.

### 18.1.3 nUpCounter

The specification is easily derivable from our specification of the 1-bit counter.

```plaintext

``
The verification is interesting in that sorting out the case structure is relatively easy, and all the hard part of the proof is concentrated in showing that the counter counts when count is enabled — essentially an arithmetic and combinational proof.

Having set the goal we induct on n and rewrite.
Base case. Both sides are already so close to each other that a simple lemma proving that \(1 - 2 \text{ EXP} (\text{SUC} 0) - 1 = 1\) and rewriting with \text{val} and \text{unbundle} are all that is required.

Inductive step. The goal that pops up is
We start in by unfolding the `let`s, removing the hidden line after first moving in the universally quantified `t`, and remove the universal quantifications.
We first focus on eliminating the relations for \( tc \). We write a special lemma for that purpose:
Rewriting with \texttt{tcLemma} enables to cancel the relations for \texttt{tc} with \texttt{CANCEL\_CONJ\_TAC}. But first we need to rearrange the association of the conjuncts on the left.

```ocaml
#let tcLemma = prove
("! n a . ((val a n = (2 \text{EXP} (SUC n)) - 1)
\\ (bv(a(SUC n)) = 1))
= (val a (SUC n) = (2 \text{EXP} (SUC(SUC n))) - 1),
REPEAT GEN\_TAC
THEN REWRITE\_TAC
[ ivals; num\_CONV "1"; GSYM (valAllOnes); GSYM(valEq); ones ];;
tcLemma =
| ~! n a.
 (val a n = (2 \text{EXP} (SUC n)) - 1) \wedge (bv(a(SUC n)) = 1) =
(val a(SUC n) = (2 \text{EXP} (SUC(SUC n))) - 1)
```

```ocaml
#o(PURE\_ONCE\_REWRITE\_TAC [ GSYM CONJ\_ASSOC ]
THEN PURE\_ONCE\_REWRITE\_TAC
[ PURE\_ONCE\_REWRITE\_RULE [ unbundle ]
 (SPECL [ "n: num"; "unbundle z t" ] tcLemma) ]
THEN CANCEL\_CONJ\_TAC);;
OK..
"(val(unbundle z(t + 1))n =
 (clr t =>
 0 |
 (clk t =>
 (ld t =>
 (val(unbundle i t)n |
 (ce t =>
 ((val(unbundle z t)n = (2 \text{EXP} (SUC n)) - 1) =>
 0 |
 (val(unbundle z t)n + 1) |
 val(unbundle z t)n)) |
 val(unbundle z t)n)))) \wedge
```
We unfold the definitions of `val` and `unbundle`.

```plaintext
#e(PURE_ONCE_BEMWRITE_TAC [ val ]
  THEN PURE_ONCE_BEMWRITE_TAC [ unbundle ]);;
OK.
"(val(unbundle z(t + 1))n =
 (clr t =>
  0 |
 (clk t =>
  (ld t =>
   val(unbundle i t)n |
   (ce t =>
    ((val(unbundle z t)n = (2 EXP (SUC n)) - 1)) =>
     (val(unbundle z t)n) + 1) |
    val(unbundle z t)n) |
   val(unbundle z t)n) |
   val(unbundle z t)n) |
   val(unbundle z t)n))" ]

() : void
```
18.1. CASE II: UP COUNTER

There seems nothing for it but to bash things out by cases. We first probe the goal with

```ml
#e(COND_CASES_TAC THEN REWRITE_TAC [ ];;

OK.
2 subgoals

% " **** case omitted **** >> %

"((val unbundle z t)n = 0) \land (bw(z(SUC n)(t + 1)) = 0) =
((val unbundle z(t + 1))n + ((2 EXP (SUC n)) * (bw(z(SUC n)(t + 1)))) = 0)"

[ "!clk clr ce ld i z tc.
   nUpCount_imp n clk clr ce ld i z tc =
   nUpCount_spec n clk clr ce ld(unbundle i)(unbundle z)tc" ]

[ "tc t =
   ce t \land (val unbundle z(t + 1))(SUC n) = (2 EXP (SUC (SUC n)) - 1)"
   ]

[ "!clk t =
  (ld t =>
   (bv(i(SUC n)t) | ((ce t \land (val unbundle z t)n = (2 EXP (SUC n)) - 1)) =>
   (bw(z(SUC n)t) = 1) => 0 | (bw(z(SUC n)t) + 1) |
   bv(z(SUC n)t))) |
   bv(z(SUC n)t))) =
((val unbundle z(t + 1))n + ((2 EXP (SUC n)) * (bw(z(SUC n)(t + 1)))) =
(clr t =>
  0 | (clk t =>
   (ld t =>
    (val unbundle i t)n + ((2 EXP (SUC n)) * (bw(i(SUC n)t))) |
    (ce t =>
     ((val unbundle z t)n + ((2 EXP (SUC n)) * (bw(z(SUC n)t)))) =
     (2 EXP (SUC n)) - 1) =>
     0 | (val unbundle z t)n + ((2 EXP (SUC n)) * (bw(z(SUC n)t))) + 1) |
     (val unbundle z t)n + ((2 EXP (SUC n)) * (bw(z(SUC n)t)))) |
     (val unbundle z(t + 1))n + ((2 EXP (SUC n)) * (bw(z(SUC n)(t + 1)))) =
     (2 EXP (SUC (SUC n)) - 1))" ]

(): void
```
We can remove this goal with a few simple rewrites:

```
# [ ADD_EQ_0; MULT_EQ_0; exp_not_0 ];
[ ! \ m \ n \ (m + n = 0) \ (m = 0) \ (n = 0); 
  [ ! \ m \ n \ (m \times n = 0) \ (m = 0) \ (n = 0); 
  [ ! \ m \ (2 \ EXP \ n = 0) ]
: thm list

#b();

% << **** trace omitted **** >> %

#(COND_CASES_TAC % *** clr *** %
  THEN REWRITE_TAC [ ADD_EQ_0; MULT_EQ_0; exp_not_0 ]);;

OK...
"(val(unbundle z(t + 1))n =
 (clk t =>
  (ld t =>
   val(unbundle i t)n |
   (ce t =>
    (val(unbundle z t)n = (2 \ EXP (SUC n)) - 1) =>
     0 |
    (val(unbundle z t)n + 1) |
    val(unbundle z t)n)) |
   val(unbundle z t)n)) /
 (bv(z(SUC n)(t + 1)) =
 (clk t =>
  (ld t =>
   bv(i(SUC n)t)) |
  ((ce t => (val(unbundle z t)n = (2 \ EXP (SUC n)) - 1)) =>
   (bv(z(SUC n)t) = 1) => 0 |
    (bv(z(SUC n)t)) + 1) |
   bv(z(SUC n)t)))) |
   bv(z(SUC n)t)) =
 (val(unbundle z(t + 1))n + ((2 \ EXP (SUC n)) \ (bv(z(SUC n)(t + 1)))) =
 (clk t =>
  (ld t =>
   (val(unbundle i t)n + ((2 \ EXP (SUC n)) \ (bv(i(SUC n)t)))) |
   (ce t =>
    (val(unbundle z t)n + ((2 \ EXP (SUC n)) \ (bv(z(SUC n)t)))) =
     (2 \ EXP (SUC(SUC n)))) - 1) =>
    0 |
    (val(unbundle z t)n + ((2 \ EXP (SUC n)) \ (bv(z(SUC n)t))) + 1) |
     (val(unbundle z t)n + ((2 \ EXP (SUC n)) \ (bv(z(SUC n)t)))) |
    (val(unbundle z t)n + ((2 \ EXP (SUC n)) \ (bv(z(SUC n)t))))" 
[ "! clk clr ce ld i z tc.
  nUpCount_imp n clk clr ce ld i z tc =
  nUpCount_spec n clk clr ce ld(unbundle i)(unbundle z)tc" ]
[ "tc t =
  ce t =>
    (val(unbundle z t)(SUC n) = (2 \ EXP (SUC(SUC n)) - 1)"
 [ "! clr t" ]
() : void
```
and have just a single goal to tackle. Let's probe this goal.

```plaintext
#e(COMB-CASES_TAC  % *** clk *** %
THRN REWRITE_TAC []);
OK.
2 subgoals
"(val(unbundle z(t + 1))n = val(unbundle z t)n) /
(bv(z(SUC n)(t + 1)) = bv(z(SUC n)t)) =
((val(unbundle z(t + 1))n + ((2 EXP (SUC n)) * (bv(z(SUC n)(t + 1)))) =
(val(unbundle z t)n + ((2 EXP (SUC n)) * (bv(z(SUC n)t))))"
[ "!clk clr ce ld i z tc.
  nUpCount Imp clk clr ce ld i z tc =
  nUpCount Spec clk clr ce ld(unbundle i)(unbundle z)tc"
][ "!tc t =
  ce t /\ (val(unbundle z t)(SUC n) = (2 EXP (SUC(SUC n))) - 1)"
][ "!clr t =
  "clk t"
][ "clk t"
]
% << ***** case omitted ***** >> %
() : void
```

This time the first sub-goal is the easier to tackle. We make use of an auxiliary lemma.

```plaintext
#valEq1;
|- !a b.
  (val a n = val b n) /\ (a(SUC n) = b(SUC n)) =
  (val a(SUC n) = val b(SUC n))
```

and then build a little package of theorems sufficient to blow away a simple generalisation of this case.

```plaintext
#let cases L
  = let th1 = SPECCL ("n:num":i) valEq1 in
  let th2 = PURE_ONCE_REWRITE_RULE [ val; unbundle ] th1 in
  let th3 = bvEq1; unbundle; th2 ];;
  cases = - : (term list -> thm list)
#
#cases [ "unbundle q (t+1)"; "unbundle q t" ];;
[|- !a b. (bv a = bv b) = (a = b);
  |- !sig t n. unbundle sig t n = sig n t;
  |- (val(unbundle q(t + 1))n = val(unbundle q t)n) /\
      (q(SUC n)(t + 1) = q(SUC n)t) =
      ((val(unbundle q(t + 1))n) +
      ((2 EXP (SUC n)) * (bv(unbundle q(t + 1)(SUC n))))) =
      (val(unbundle q t)n) +
      ((2 EXP (SUC n)) * (bv(unbundle q t(SUC n))))]
  : thm list
```
We back up and try our luck.

```c
#b();

% << **** trace omitted **** >> %

#e(COND_CASES_TAC % *** clk *** %
THEN REWRITE_TAC {cases [ "unbundle q (t+1)"; "unbundle q t" ]};

OK...

"(val(unbundle z(t+1))n =
  (ld t =>
    val(unbundle i t)n |
    (ce t =>
      ((val(unbundle z t)n = (2 EXP (SUC n)) - 1) =>
        0 |
      (val(unbundle z t)n + 1) |
      val(unbundle z t)n))) |
  (bv(z(SUC n)(t+1)) =
    (ld t =>
      bv(i(SUC n)t) |
      ((ce t \ (val(unbundle z t)n = (2 EXP (SUC n)) - 1)) =>
        ((bv(z(SUC n)t) = 1) => 0 | (bv(z(SUC n)t)) + 1) |
      bv(z(SUC n)t)))) =
  ((val(unbundle z(t+1))n + ((2 EXP (SUC n)) * (bv(z(SUC n)(t+1)))) =
    (ld t =>
      (val(unbundle i t)n + ((2 EXP (SUC n)) * (bv(i(SUC n)t))) |
      (ce t =>
        ((val(unbundle z t)n) + ((2 EXP (SUC n)) * (bv(z(SUC n)t)))) =
        (2 EXP (SUC(SUC n)) - 1) =>
        0 |
      (val(unbundle z t)n) + ((2 EXP (SUC n)) * (bv(z(SUC n)t))) + 1) |
      (val(unbundle z t)n) + ((2 EXP (SUC n)) * (bv(z(SUC n)t))))))"
```

Not only does rewriting with `cases` work here, it also works on the next two cases (which we found out by probing with `COND_CASES_TAC` then `REWRITE_TAC` first).
We are now left with the last case when clear is low and clk t ∧ ~ld t ∧ ce t, i.e., we increment. We leave this as an exercise (we called our lemma ceCases and found it took over one page of HOL). Here is our final theorem in tidy form:
#let nUpCount_correct = prove_thm
('nUpCount_correct',
"! n clk clr ce ld i z tc .
  nUpCount_imp n clk clr ce ld i z tc
  = nUpCount_spec n clk clr ce ld (unbundle i) (unbundle z) tc",
INDUCT_TAC THEN REPEAT GEN_TAC
THEN PURE_ASM_REWRITE_TAC
[ nUpCount_imp; nUpCount_spec;
  UpCount_correct; UpCount_spec
]
THENL
[ PURE_REWRITE_TAC [ val; unbundle; expSubl; num_CONV "1" ]
  THEN REFL_TAC ;
  \% remove hidden lines and foralls \%
  let_TAC
  THEN FORALL_AND_TAC THEN EXISTS_ELIM_TAC
  THEN AND_FORALL_TAC THEN FORALL_EQ_TAC
  \% remove the equations for tc \%
  THEN PURE_ONCE_REWRITE_TAC [ GSYM CONJ_ASSOC ]
  THEN PURE_ONCE_REWRITE_TAC
[ PURE_ONCE_REWRITE_RULE [ unbundle ]
  (SPECL [ "n":num; "unbundle z t" ] tcLemma)]
  THEN CANCEL_CONJ_TAC
  THEN PURE_REWRITE_TAC [ val; unbundle ]
  THEN COND_CASES_TAC \% \*\*\* clr \*\*\* \%
  THEN REWRITE_TAC [ ADD_EQ_0; MULT_EQ_0; exp_not_0 ]
  THEN COND_CASES_TAC \% \*\*\* clk \*\*\* \%
  THEN REWRITE_TAC (cases [ "unbundle q (t+1)"; "unbundle q t" ])
  THEN COND_CASES_TAC \% \*\*\* ld \*\*\* \%
  THEN REWRITE_TAC (cases [ "unbundle q (t+1)"; "unbundle i t" ])
  THEN COND_CASES_TAC \% \*\*\* ce \*\*\* \%
  THEN ASM_REWRITE_TAC (cases [ "unbundle q (t+1)"; "unbundle q t" ])
  THEN ceCases 
);)

nUpCount_correct =
! n clk clr ce ld i z tc .
  nUpCount_imp n clk clr ce ld i z tc
  = nUpCount_spec n clk clr ce ld (unbundle i) (unbundle z) tc

Run time: 22.3s
Garbage collection time: 29.4s
Intermediate theorems generated: 5140
Part VII

Recursive data types in HOL
Chapter 19

Recursive data types

Recursive data types and functions over them are amongst the jewels of modern programming languages. In this chapter we introduce and apply the recursive data types package of which was designed and implemented by Tom Melham. For a restricted (but still useful) class of data types, Tom found it possible to define the new types in terms of existing types and then derive properties about the new type by formal proof. This guarantees that adding the new type to the logic cannot introduce any inconsistencies. A good deal of effort and care was taken to ensure that the package was reasonably efficient. [77, 78, 79] are recommended reading for his implementation approach and philosophy.

In chapter 19 we show you how to define your own data structures and functions over them in HOL. If you are not used to using a modern functional programming language, we suggest you take time out and read the wonderful texts by Burge [16] and/or Bird and Wadler [8], or David Turner’s papers [108, 109, 110]. We illustrate the basic approach using our own definition of a binary tree, and then introduce the built-in theory of lists before applying it to prove some facts about insert sort.

In chapter 20, we prove the correctness of the compiler for a very small language running on a very small computer. You will have to read chapter 20 to realise how small we mean by “very small”. Nonetheless it puts across the basic idea. Realistic examples can be found in [101].

Chapter 21 contains a much more substantial example on finite state machines.

19.1 Tools for defining data types

In this section we show the standard steps in defining a recursive datatype using binary trees as our model. The first step is to define the data structure in which we are interested. This is accomplished by calling \texttt{define_type} which takes two string arguments. The first is an identifier under which the data type will be saved in the current theory. The second is string which defines the data type.
Proving basic facts about a data type. It is usual to proceed by proving several useful basic theorems from the data type axiom using supplied tools. Each of them takes a single argument and returns a single theorem. We show them in action one by one.

```ocaml
#let TREE_INDUCT = prove_induction_thm TREE_AXIOM;;
TREE_INDUCT =
|→ !P.
  P EMPTY /
  (?x. P (LEAF x)) /
  (?T1 T2. P T1 /\ P T2 ==> P (NODE T1 T2)) ==> (?T. P T)
```

proves the induction theorem over trees, namely that if a proposition holds over (i) the empty tree, (ii) for each LEAF x, (iii) and from the assumption that it holds for left (T1) and right (T2) subtrees we can show that it holds for the NODE T1 T2, then it holds for every tree.

```ocaml
#let TREE_11 = prove_constructors_one_one TREE_AXIOM;;
TREE_11 =
|→ (?x x'. (LEAF x = LEAF x') = (x = x')) /
  (?T1 T2 T1' T2'.
    (NODE T1 T2 = NODE T1' T2') = (T1 = T1') /\ (T2 = T2'))
```

proves that two leaves the same only if their contents are equal, and that two nodes are the same only if their left subtrees are equal and their right subtrees are equal.
proves that the constructors yield distinct values.

```ml
let TREE_CASES = prove_cases_thm TREE_INDUCT;;
TREE_CASES = |
| "T". (T' = EMPTY) \ (\x. T' = LEAF x) \ (T1 T2. T' = NODE T1 T2)
```

derives the tree cases from the induction theorem. Finally we construct an appropriate tactic from TREE_INDUCT using the tactical INDUCT_THEN.

```ml
let TREE_INDUCT_TAC = INDUCT_THEN TREE_INDUCT_ASSUME_TAC;;
TREE_INDUCT_TAC = - : tactic
```

Once these basic theorems are established, we can start to define useful functions over trees and prove properties about them.

**Doing it all in one step.** It is a simple matter to gather together the generation of all five basic theorems and the generation of the associated tactic into one simple ML function which takes two strings as arguments — the very strings you would pass to a call on `define_type`.

```ml
let axiomatiseDataType stem defn
    = let axiom = define_type stem defn
      in
      let induct = prove_induction_thm axiom
      and one_one = prove_constructors_one_one axiom
      and dist = prove_constructors_distinct axiom
      in
      let cases = prove_cases_thm induct
      and induct_tac = INDUCT_THEN induct_ASSUME_TAC
      in
      let (axiom, induct, one_one, dist, cases, induct_tac) =
      (axiom, induct, one_one, dist, cases, induct_tac);
      axiomatiseDataType =
      -
      : (string -> string -> (thm # thm # thm # thm # tactic))
```

A call on `axiomatiseDataType` returns a 6-tuple of 5 theorems and one tactic. Here it is in action on a datatype for the natural numbers:

```ml
let (NUM_AXIOM, NUM_INDUCT, NUM_11, NUM_DISTINCT, NUM_CASES, NUM_INDUCT_TAC) =
axiomatiseDataType

'NUM' |
NUM_AXIOM = |- f e f. (?f. fn. (fn ZERO = e) \ (\n. fn(SUC n) = f(fn n)))
NUM_INDUCT = |- !P. P ZERO \ (\n. P n == P(SUC n)) ==> (\n. P n)
NUM_11 = |- !n. n.(SUC n = n) == (n = n')
NUM_DISTINCT = |- !n. n.(ZERO = SUC n) == (n = n')
NUM_CASES = |- !n. (n = ZERO) \ (\n. n = SUC n')
NUM_INDUCT_TAC = - : tactic
```
Note that the clause `dist = prove_constructors_distinct axiom` will fail if the data type has only one constructor:

```ocaml
#let (U_AXIOM, U_INDUCT, U_ONE_ONE, U_DISTINCT, U_CASES, U_INDUCT_TAC) = axiomiseDataType
  'U
  'U = UDATA num';
evaluation failed prove_constructors_distinct: invalid input
```

We can make `axiomiseDataType` secure against this sort of failure by trapping the exception and returning a “harmless” default theorem, say `TRUTH = |- T`, as in

```ocaml
#let axiomiseDataType stem defn =
  let axiom = define_type stem defn in
  let induct = prove_induction_thm axiom
  and one_one = prove_constructors_one_one axiom
  and dist = (prove_constructors_distinct axiom) ? TRUTH in
  let cases = prove_cases_thm induct
  and induct_tac = INDUCT_THEN induct ASSUME_TAC in
  (axiom, induct, one_one, dist, cases, induct_tac);;
axiomiseDataType =
  -
  : (string -> string -> (thm # thm # thm # thm # tactic))
```

### Defining functions over data types.

To define a function over a data type we use the built-in function `new_recursive_definition` which takes four arguments (remember that `conv` is short for `term -> thm`):

1. a bool which is `true` if the operation being defined is infix; `false` if prefix. We only use the infix option.
2. the axiom for the data type over which we are making our definition
3. the theory “save” name for this function
4. a term defining the function by cases of the constructors of the data type

and it returns the definition as a theorem. We now venture forth and give definitions of REFLECT, the reflection of a tree of arbitrary type,

```ocaml
#let REFLECT = new_recursive_definition
false % infix or not %
TREE_AXIOM 'REFLECT'
"(REFLECT EMPTY = EMPTY) /
(REFLECT (LEAF (n : *)) = (LEAF n)) /
(REFLECT (NODE (L : * TREE) (R : * TREE))
  = (NODE (REFLECT R) (REFLECT L)))
";;
REFLECT = ...
```

and LEAVES, which counts the number of leaves in a tree.

```ocaml
#let LEAVES = new_recursive_definition
false
TREE_AXIOM 'LEAVES'
"(LEAVES EMPTY = 0) /
(LEAVES (LEAF (n : *)) = 1) /
(LEAVES (NODE L R) = (LEAVES L) + (LEAVES R))
";;
LEAVES = ...
```

**Example 19.1.1 REFLECT_TWICE**

Prove ⊢ ! t. REFLECT(REFLECT t) = t

When setting the goal we are not allowed to forget that trees are of arbitrary type:

```ocaml
#g "! t . LEAVES(REFLECT t) = LEAVES t";;
Indeterminate types: "#:(TREES -> bool) -> bool"
evaluation failed types indeterminate in quotation
```

We have to instantiate t.
CHAPTER 19. RECURSIVE DATA TYPES

The first step is to induct over the structure of trees.

We state the obvious: three cases emerge, one for each constructor. The case for the constructor NODE has two assumptions, one for the left subtree, and one for the right subtree. As ever the base cases are trivial:

```latex
\texttt{\#e(TREE\_INDUCT\_TAC);;}
\texttt{OK.}
\texttt{3 subgoals}
\texttt{"REFLECT(REFLECT(NODE t t')) = NODE t t'"}
\texttt{[ "REFLECT(REFLECT t) = t" ]}
\texttt{[ "REFLECT(REFLECT t') = t''" ]}
\texttt{"!x. REFLECT(REFLECT(LEAF x)) = LEAF x"}
\texttt{"REFLECT(REFLECT EMPTY) = EMPTY"}
\texttt{() : void}

\texttt{\#e(REWRITE\_TAC \{ REFLECT \};;}
\texttt{OK.}
\texttt{goal proved}
\texttt{|- REFLECT(REFLECT EMPTY) = EMPTY}

Previous subproof:
2 subgoals
\texttt{"REFLECT(REFLECT(NODE t t')) = NODE t t'"}
\texttt{[ "REFLECT(REFLECT t) = t" ]}
\texttt{[ "REFLECT(REFLECT t') = t''" ]}
\texttt{"!x. REFLECT(REFLECT(LEAF x)) = LEAF x"}
\texttt{() : void}
```
and so is the third. We need but rewrite with \texttt{REFLECT} and then use the assumptions.

To make the presentation of the theorem a little more tidy, we start by calling \texttt{TREE\_INDUCT\_TAC THEN REPEAT GEN\_TAC} which, after casing over the structure of trees, removes any generalisations from the subgoals (and has no effect should they be free of generalisations).

```lean
#let REFLECT\_TWICE = prove_thm
  ("REFLECT\_TWICE", "! (t: \textsc{tree}). \texttt{REFLECT(REFLECT t)} = t",
   TREE\_INDUCT\_TAC THEN REPEAT GEN\_TAC
   THEN ASM\_REWRITE\_TAC [ REFLECT ]);;
REFLECT\_TWICE = ! t. \texttt{REFLECT(REFLECT t)} = t
Run time: 0.5s
Intermediate theorems generated: 119
```
CHAPTER 19. RECURSIVE DATA TYPES

Example 19.1.2 LEAF_REFLSAME
Prove \(\forall t. \text{LEAVES(REFLECT } t) = \text{LEAVES } t\)
As a second trivial theorem we prove that reflecting a tree of numbers does not alter the sum: In this case, we specialise the tree type to be \(\texttt{:num TREE}\).

```
#g "\(t: \texttt{num TREE}\). \text{LEAVES(REFLECT } t) = \text{LEAVES } t\"
"\(t. \text{LEAVES(REFLECT } t) = \text{LEAVES } t\"
()
```

induct, strip away any subgoal generalisations and rewrite with obvious theorems (and the assumption list). Only one subgoal remains:

```
#e(TREE_INDUCT_TAC THEN REPEAT GEN_TAC
    THEN ASM_REWRITE_TAC [ LEAVES; REFLECT ]);
OK.
"\(\text{LEAVES } t\)' + (\text{LEAVES } t) = (\text{LEAVES } t) + (\text{LEAVES } t)'"
    [ "\text{LEAVES(REFLECT } t) = \text{LEAVES } t" ]
    [ "\text{LEAVES(REFLECT } t') = \text{LEAVES } t'" ]
()
```

which can be cleared by substituting with ADD_SYM

```
#ADD_SYM;;
|- \!m n. m + n = n + m
```

and then calling REFL_TAC (rewriting with ADD_SYM will of course loop).

```
#e(SUBST1_TAC
    (SPEC \("(\text{LEAVES: num TREE->num}) t\)";
      "(\text{LEAVES: num TREE->num}) t"
    ] ADD_SYM)
    THEN REFL_TAC);
OK.
goal proved
.. |- \(\text{LEAVES } t\)' + (\text{LEAVES } t) = (\text{LEAVES } t) + (\text{LEAVES } t)'
  |- \!t. \text{LEAVES(REFLECT } t) = \text{LEAVES } t
Previous subproof:
goal proved
()
```

The proof in tidy form reads:
#let LEAF_REFL_SYM = prove_thm
( "!(t: TREE). LEAVES(REFLECT t) = LEAVES t",
 TREE_INDUCT_TAC THEN REPEAT GEN_TAC
 THEN ASM_REWRITE_TAC [ LEAVES; REFLECT ]
 THEN SUBST_TAC
 (SPECL ["(LEAVES:* TREE->num) t"; "(LEAVES:* TREE->num) t" ] ADD_SYM)
 THEN REFL_TAC);

LEAF_REFL_SYM = |- !t. LEAVES(REFLECT t) = LEAVES t

Run time: 0.7s

Intermediate theorems generated: 164
19.2 The built-in list datatype

HOL contains a pre-defined theory for lists. A listing is given as appendix ???. Here are some selected definitions and theorems:

**Definitions:**
- **NULL_DEF**: |- (NULL [] = T) /
  \! (h t. NULL(CONS h t) = F)
- **HD**: |- \! h t. HD(CONS h t) = h
- **TL**: |- \! h t. TL(CONS h t) = t
- **APPEND**: 
  |- (!l1 12 h. APPEND(CONS h l1) l2 = CONS h(APPEND l1 l2))
- **LENGTH**: 
  |- (LENGTH [] = 0) /
  \! (h t. LENGTH(CONS h t) = SUC(LENGTH t))
- **MAP**: 
  |- (!f MAP f[] = []) /
  \! (h t. MAP f(CONS h t) = CONS(f h)(MAP f t))

**Theorems:**
- **list_Axiom**: 
  |- !x f. ?! fn. (fn [] = x) /
  \! (h t. fn(CONS h t) = f(fn t)h t)
- **list_INDUCT**: 
  |- !P. P [] /
  \! (!? P t \rightarrow \! (?! t. P(CONS h t))) \rightarrow \! (!! P 1)
- **CONS_I1**: 
  |- !! h t h' t'. (CONS h t = CONS h' t') \rightarrow \! (h = h') /
  \! (t = t')
- **NOT_NULL_CONS**: 
  |- \! h t. \~(CONS h t)
- **NOT_CONS_NULL**: 
  |- \! h t. \~(CONS h t = [])
- **NOT_EQ_LIST**: 
  |- !h1 h2. \~(h1 = h2) \rightarrow \! (!! 11 12. \~(CONS h1 l1 = CONS h2 l2))
- **EQ_LIST**: 
  |- !h1 h2. 
  \! (h1 = h2) \rightarrow \! (!! 11 12. \! (l1 = l2) \rightarrow \! (CONS h1 l1 = CONS h2 l2))
- **CONS_I1**: 
  |- \! ! h. \~NULL l \rightarrow \! (CONS(h1 l1 = CONS h2 l2))
- **APPEND_ASSOC**: 
  |- !11 12 13. APPEND 11(APPEND 12 13) = APPEND(APPEND 11 12)13

**Tactic:**

**LIST_INDUCT_TAC**

The empty list is denoted by []; a nonempty list by CONS h L. Remember list_Axiom and LIST_INDUCT_TAC — they are used to make recursive definitions over lists and carry out induction over lists respectively.

Since lists are very versatile, it will pay you to familiarise yourself with this theory (the version current at the time of going to press is in appendix ???. We expect it will grow.) Here we define the filter operation over lists and prove a useful theorem. You are asked to prove more facts in the exercises.
Example 19.2.3 FILTER_TWICE

Given the definition

```lisp
#list_Axiom;;
| ! x f. ?! fn. (fn[] = x) /\ (!h t. fn(CONS h t) = f(fn t)h t)

#let filter
  = new_recursive_definition
    false
  list_Axiom
  "filter"
    (filter (p:-->bool) [] = []) /\  
    (filter p (CONS a L))  
      = (p a) => CONS a (filter p L)  
        | filter p L
    ";
  filter = ...
```

prove that ! L p q . (filter p (filter q L) = filter (\ x . (p x \ q x)) L)

We start by setting the goal, and inducting on the list variable L.

```lisp
# " ! (L: list) (p:-->bool) (q:-->bool).
  filter p (filter q L)  
    = filter (\ x . (p x \ (q x)) L)

"! L p q. filter p (filter q L) = filter (\ x . (p x \ q x)) L"

() : void

#LIST_INDUCT_TAC;;
- : Tactic

#e(LIST_INDUCT_TAC);;
OK...
2 subgoals
  "! h p q. filter p (filter q (CONS h L)) = filter (\ x . p x \ q x) (CONS h L)"
    [ "! p q. filter p (filter q L) = filter (\ x . p x \ q x) L"
      ]

  "! p q. filter p (filter q[]) = filter (\ x . p x \ q x) []"

() : void
```

As expected, two cases are generated, one for the empty list [], the other for non-empty lists CONS h L. Notice that the introduced variable h has been generalised in the subgoal for the nonempty list case. Notice also that by ordering the variables L p q in the goal, we have the assumption

"! p q. filter p (filter q L) = filter (\ x . p x \ q x) L"

for the non-empty case. Had we generalised the variables in the order p q L instead, the assumption would have read
"filter p(filter q L) = filter(\x. p x \ q x)\"L"

which is specific to p and q and thus less useful. A seemingly minor point, but one that simplifies many proofs.

We backup, and this time strip away the generalisations on the subgoals and rewrite from the assumptions with filter:

```
#b();
"!L p q. filter p(filter q L) = filter(\x. p x \ q x)\"L"
()
#c(LIST_INDUCT_TAC THEN REPEAT GEN_TAC
  THEN ASM_REWRITE_TAC [ filter ]);;
OK..
"filter p(q h \ CONS h(filter q L) \ filter q L) =
  (\x. p x \ q x)h \ CONS h(filter(\x. p x /\ q x)L) \ filter(\x. p x /\ q x)L)"
  [
  "!p q. filter p(filter q L) = filter(\x. p x /\ q x)L"
  ]
()
```

We next apply a \(\beta\) reduction.

```
#e(CONV_TAC(CHANGED_CONV(DEPTH_CONV BETA_CONV))));;
OK..
"filter p(q h \ CONS h(filter q L) \ filter q L) =
  (p h /\ q h) \ CONS h(filter(\x. p x /\ q x)L) \ filter(\x. p x /\ q x)L)"
  [
  "!p q. filter p(filter q L) = filter(\x. p x /\ q x)L"
  ]
()
```

We now have if-then-else structures on the left and on the right. The easiest way is to get rid of them altogether by casing on \(p \ h\) and on \(q \ h\). Looking ahead a little, when \(q \ h = T\), this will give rise to a sub-term \(filter p (CONS h (filter q L))\) which we can further simplify by rewriting with filter. When \(q \ h = F\), we get a term of the form \(filter p (filter q L)\) which we can rewrite from the assumptions.
Two cases vanish, which is nice. But for the other two, we are left with conditions on \( p h \) which terms appeared only after the bool casing had been applied. We have done the case split, but we need to remember which case is which. If this information were available (say on the assumption list) we could rewrite with it and be left with either a tautology (when \( p h = T \)) or a simple rewrite from the assumption list (when \( p h = F \)). We backup and try again with ASM\_CASES\_TAC.
Here is the theorem in tidy form.

```plaintext
#let filter_TWICE = prove_thm
(f"filter_TWICE",
  " ! L (p:bool) (q:bool).
   filter p (filter q L)
   = filter (\x . (p x) \& (q x)) L",
  LIST_INDUCT_TAC THEN REPEAT GEN_TAC
  THEN ASM_REWRITE_TAC [ filter ]
  THEN CONV_TAC(CHANGED_CONV(DEPTH_CONV BETA_CONV))
  THEN MAP_EVERY ASM_CASES_TAC [ "(p:bool) h"; "(q:bool) h" ]
  THEN ASM_REWRITE_TAC [ filter ]);
filter_TWICE = |- ! L p q. filter p(filter q L) = filter(\x. p x \& q x)L

Run time: 2.1s
Intermediate theorems generated: 350
```
EXERCISES 19

Exercise 19.1  Using the given definition of the NUM data type for the natural numbers, prove the theorems:

1. |- ! a . NUM_ADD ZERO a = a
2. |- ! a . NUM_ADD a ZERO = a
3. |- ! a b . NUM_ADD (SUCC a) b = SUCC(NUM_ADD a b)
4. |- ! a b . NUM_ADD a (SUCC b) = SUCC(NUM_ADD a b)

Exercise 19.2  Prove the following theorems over lists:

1. |- ! L a . APPEND (CONS a []) L = CONS a L
2. |- ! L . (APPEND L [] = L)
3. Given that circle f g x = f(g x), prove that
   |- ! L f g . ((MAP f) circle (MAP g)) L = (MAP (circle f g) L)
4. Prove that
   |- ! L p q . filter p (filter q L) = filter q (filter p L)

Exercise 19.3  Given the definition

let FRONTIER
  = new_recursive_definition
    false
  TREE_AXIOM
  'FRONTIER'
  "(FRONTIER EMPTY = []) /
   (FRONTIER (LEAF n:*)) = [ n ] /
   (FRONTIER (NODE L R)) = APPEND (FRONTIER L) (FRONTIER R))
";

prove that ! t . LEAVES t = LENGTH(FRONTIER t).
Example 19.2.4 Insert sort
As a reasonably sized example over lists, we now define the notions of a sorted list of numbers and of insert sort over a list of numbers.

sorted $L$ is defined by induction over the structure of $L$:

- $\text{sorted } [] : \text{ trivially true }$
- $\text{sorted } (\text{CONS } a \ L) : \text{ trivially true if } L = [] \text{ when we return the singleton list } [a]. \text{ Otherwise, } L \text{ must be sorted and we require that } a \leq \text{ HD } L.$

We define the operation of inserting a new number $a$ into an already sorted list $L$, \text{insert } $a \ L$, by induction over $L$ as:

- $\text{insert } a \ [] : [a], \text{ a list with one item. }$
- $\text{insert } a \ (\text{CONS } h \ L) : \text{ there are two cases to consider: }$
  1. $a \leq h: \text{ CONS } a \ (\text{CONS } h \ L), \text{ just tack } a \text{ onto the front. }$
  2. $a > h: \text{ CONS } h \ (\text{insert } a \ L), \text{ we insert } a \text{ into the tail of the list after } h.$

We can now define insert sort, \text{SORT } $L$, again by induction over $L$:

- $\text{SORT } [] : \text{ trivial } — [].$
- $\text{SORT}(\text{CONS } a \ L) : \text{insert } a \ (\text{SORT } L), \text{ sort the tail of the list } L, \text{ and then insert } a \text{ into it.}$

From these definitions, we seek to prove the following four properties of insert sort:

(i) $\forall L . \text{ sorted } (\text{SORT } L)$
(ii) $\forall x . x \in \text{SORT } L = x \in L$
(iii) $\forall x . \text{LENGTH}(\text{SORT } L) = \text{LENGTH } L$
(iv) $\forall x . \text{oocs } x \ (\text{SORT } L) = \text{oocs } x \ L$

where (iv) makes sure that each occurrence of $x$ in $L$ will be mapped into \text{SORT } $L$. 
Proof

We start by logging in, opening a new theory, and reminding ourselves of certain key definitions:

```
#new_theory 'insert';
() : void

#list_Axiom;
|- 'x f. ?! fn. (fn[] = x) /
  (!h t. fn(CONS h t) = f(fn t)h t)

#list_INDUCT;
|- !P. P[] /
  (!!t. P t => (!h. P(CONS h t))) => (!1. P 1)

#LIST_INDUCT_TAC;
- : tactic
```

We then enter the basic definitions. First sorted:

```
#let sorted = new_recursive_definition
false
list_Axiom
'sorted'
"(sorted [] = T) /
  (sorted (CONS h L) = ((L = []) \ (h <= (HD L))) \ (sorted L))";;

sorted = ...

#let insert = new_recursive_definition
false
list_Axiom
'insert'
"((insert (a:num) [] = (CONS a [])) /
  ((insert a (CONS (h:num) L))
   = ((a <= h) => (CONS a (CONS h L)))
     | (CONS h (insert a L)))");;

insert = ...

#let SORT = new_recursive_definition
false
list_Axiom
'SORT'
"((SORT [] = []) /
  (SORT (CONS h L) = (insert h (SORT L)))");;

SORT = ...
```

We also require definitions of isMem and occs (occurrences), both of which are straightforward:
CHAPTER 19. RECURSIVE DATA TYPES

```plaintext
#let isMem = new_recursive_definition
false
list_Axiom
'isMem'
"(isMem (x:num) [] = F) /\ 
 (isMem x (CONS h L) = ((x = h) /\ (isMem x L)))";;
isMem = ...

#let occs = new_recursive_definition
false
list_Axiom
'occs'
"(occs (x:num) [] = 0) /\ 
 (occs x (CONS a L) = ((x=\) => SUC(occs x L) | occs x L))
";;
occs = ...
```

(i) ⊢ ! L. sorted(SORT L)

We start this proof by probing the end theorem and letting the lemmas we will need to complete it reveal themselves. The first step is obvious:

```plaintext
#g "! L. sorted(SORT L))";;
"!L. sorted(SORT L)"

() : void
```

We are stuck but what we have suggests we try and prove

```
|- ! L x. sorted L => sorted (insert x L)
```

as a separate lemma. We set a fresh goal and probe again, inducting over the list L, stripping away other variables in subgoals, and doing the obvious rewrites.
19.2. THE BUILT-IN LIST DATATYPE

```plaintext
#e "(" x. (sorted L) => (sorted (insert x L)))";;
"(" x. sorted L => sorted (insert x L))

() : void

#e (LIST_INDUCT_TAC THEN REPEAT GEN_TAC
   THEN REWRITE_TAC [sorted; insert]);;
OK.
"((L = []) \ / h <= (HD L)) \ / sorted L =>
 sorted (x <= h => CONS x(CONS h L) | CONS h(insert x L))"
   ["x. sorted L => sorted (insert x L)"

() : void

The base case has vanished. If we bool-case on x <= h, the right hand side will collapse into either

- sorted(CONS x(CONS h L)), or

- sorted(CONS h(insert x L))

both of which can be rewritten with sorted

```plaintext
#e (ASM_CASES_TAC "x <= h"
   THEN ASM_REWRITE_TAC [sorted]);;
OK.
2 subgoals
"((L = []) \ / h <= (HD L)) \ / sorted L =>
((insert x L = []) \ / h <= (HD(insert x L))) \ / sorted(insert x L)"
   ["x. sorted L => sorted(insert x L)"
   ["x <= h"
"((L = []) \ / h <= (HD L)) \ / sorted L =>
((CONS h L = []) \ / x <= (HD(CONS h L))) /
((L = []) \ / h <= (HD L)) /
sorted L"
   ["x. sorted L => sorted(insert x L)"
   ["x <= h"

() : void

The top goal is now very easy. The right hand term in

(CONS h L = []) \ / x <= (HD(CONS h L))

reduces to x <= h when we rewrite with HD. Since x <= h is on the assumption list, we can rewrite this goal into an implication where the antecedent and the implicand are identical. We backup and try again:
CHAPTER 19. RECURSIVE DATA TYPES

We are now down to one case. Stripping the goal puts \texttt{sorted L} onto the assumption list, and an application of \texttt{RES_TAC} will then add (at least) \texttt{sorted(insert x L)}.
Two cases arise since we had a disjunction in the antecedent. In the top subgoal, \([x] = []\) (i.e. \textsc{cons} \(x \ [\] = [\])\) is certainly false; and so is \texttt{insert x L} \(= []\) in the other subgoal. This leaves us to prove \(h <= x\) and \(h <= \texttt{HD(insert x L)}\) from their respective assumptions.

The plan of campaign is as follows:

\begin{itemize}
  \item **top goal**: we show that \(! m n \ . \ (m <= n) ==> (n <= m)\). Given the assumption \("x <= h\), applying \texttt{IMP_RES_TAC} to this lemma puts \(h <= x\) on the assumption list.
  \item **bottom goal**: we have to show that \(h <= (\texttt{HD(insert x L)})\) The assumptions \("x <= h\), \(h <= (\texttt{HD L})\) and \texttt{sorted L} are sufficient. We prove an appropriate lemma.
\end{itemize}

**Top goal**

```haskell
#NOT_LESS;;
\[ \neg ! m n . \ (m <= n) =\Rightarrow (n <= m) \]

#LESS_OR_EQ;;
\[ \neg ! m n . \ m <= n \Rightarrow n <= m \ \text{\&} (m = n) \]

let triv1 = prove
("! m n . \ (m <= n) ==> (n <= m)\),
REPEAT GEN_TAC
THEN SUBST1_TAC (SPEC_ALL LESS_OR_EQ)
THEN PURE_REWRITE_TAC \[ DE_MORGAN_THM; NOT_LESS \]
THEN STRIP_TAC;;
triv1 = \[ \neg ! m n . \ (m <= n) ==> n <= m \]
```

**Bottom goal**

We prove the following lemma:

```haskell
#"! L x h . \ (h <= x) \ /\ (h <= HD L) \ /\ (sorted L))
    ==> (h <= HD(insert x L))";;

"! L x h . \ h <= x \ /\ h <= (HD L) \ /\ sorted L ==\Rightarrow h <= (HD(insert x L))"
```

() : void

We start by inducting on \texttt{L}, stripping away other generalisations, rewriting, and then pushing the left hand side of the goal onto the assumption list.
the same subgoal but with slightly different assumptions. Clearly a case split on \( x \leq h \) is called for. When we do that, the if-then-else structures collapse and we are left with only two distinct subgoals:

- \( h' \leq (\text{HD}(\text{CONS } x(\text{CONS } h \ L))) = h' \leq x \),
- \( h' \leq \text{HD}(\text{CONS } h(\text{insert } x \ L))) = h' = h \)

both of which are assumptions. We backup and complete the proof.

Here is the tidy proof:
The remaining two steps in this little proof hierarchy are trivial:

(`! `L x. `h `< `x /\ `h `< `(HD `L) /\ (`sorted `L) `=> `(HD (`insert `x `L)))`.

Once again we start the proof by setting the goal and probing to see what turns up.
What we have to prove is reasonable: we rewrite the goal by rewriting with the mirror of the assumption. Remember `ASSUM_LIST tac` hands over a copy of the assumption list to `tac`. Our `tac` rotates the one assumption and then rewrites `isMem x L`.

This suggests that we set another goal and prove the following auxiliary lemma: The first steps are obvious:

We now simplify the structure on the right by bool-casing on `h' <= h` which enables us to rewrite with `isMem` on the right leaving:

- case `(h' <= h) = T : rhs = isMem x (CONS h' (CONS h L))` which simplifies first to `(x = h') \ isMem x (CONS h L)` and then to `(x = h') \ isMem x L` which is identical the term on the left.
• case \((h' \leq h) = F: \text{rhs} = \text{isMem x(CONS h(insert h' L))}\) which simplifies to \((x = h) \lor \text{isMem x(insert h' L)}\).

```
#e(B001.CASES_TAC "h' <= h"
THEN REWRITE_TAC [ isMem ]);;
OK.
"(x = h') \lor (x = h) \lor \text{isMem x L} = (x = h) \lor \text{isMem x(insert h' L)}"
[ "x h. (x = h) \lor \text{isMem x L} = \text{isMem x(insert h L)}"
]
() : void
```

All that is needed now is to bool-case on \(x = h\) and rewrite from the assumptions.

```
#e(B001.CASES_TAC "x=h:num"
THEN ASM_REWRITE_TAC [ ];);
OK.
goal proved
. |- (x = h') \lor (x = h) \lor \text{isMem x L} = (x = h) \lor \text{isMem x(insert h' L)}
. |- (x = h') \lor (x = h) \lor \text{isMem x L} =
   \text{isMem x(h' <= h => CONS h'(CONS h L) | CONS h(insert h' L))}
. |- !L x h. (x = h) \lor \text{isMem x L} = \text{isMem x(insert h L)}

Previous subproof:
goal proved
() : void
```

Here are the two lemmas in tidy form.

```
#let lem4 = prove
("'! L x . isMem x (SORT L) = isMem x L",
 LIST_INDUCT_TAC THEN REPEAT GEN_TAC
 THEN REWRITE_TAC [ insert; isMem ]
 THEN B001.CASES_TAC "h' <= h"
 THEN REWRITE_TAC [ isMem ]
 THEN B001.CASES_TAC "x=h:num"
 THEN ASM_REWRITE_TAC [ ];);
lem4 = |- !L x h. (x = h) \lor \text{isMem x L} = \text{isMem x(insert h L)}
Run time: 0.9s
Intermediate theorems generated: 176
```

```
#let lem5 = prove
("'! L x . isMem x (SORT L) = isMem x L",
 LIST_INDUCT_TAC THEN REPEAT GEN_TAC
 THEN REWRITE_TAC [ isMem; SORT ]
 THEN ASSUM_LIST(PURE_ONCE_REWRITE_TAC o (map GSYM))
 THEN REWRITE_TAC [ lem4 ]);;
lem5 = |- !L x . isMem x(SORT L) = isMem x L
Run time: 0.4s
Intermediate theorems generated: 101
```
\( (iii) \vdash \lambda (L:\text{num list}). \text{LENGTH}(\text{SORT} \ L) = \text{LENGTH} \ L \)

Once again we attack the goal in the obvious way:

\[
\begin{align*}
\text{#} \text{LENGTH} &; ; \\
\vdash (\text{LENGTH}[] = 0 \ \& \ (\forall h \ t. \ \text{LENGTH}(\text{CONS} \ h \ t) = \text{SUC}(\text{LENGTH} \ t)) \\
\text{#g} \ "(L:\text{num list}). \text{LENGTH}(\text{SORT} \ L) = \text{LENGTH} \ L" ; ; \\
"(L. \ \text{LENGTH}(\text{SORT} \ L) = \text{LENGTH} \ L" \\
() : \text{void} \\
\text{#e} (\text{LIST\_INDUCT\_TAC\ \text{THEN\ \text{REPEAT\ \text{GEN\_TAC}}}} \\
\text{\hspace{1cm} \text{THEN\ \text{ASM\_REWRITE\_TAC [ \text{SORT}; \text{LENGTH} ]}}) ; ; \\
\text{OK} . . \\
"\text{LENGTH}(\text{insert} \ h(\text{SORT} \ L)) = \text{SUC}(\text{LENGTH} \ L)"
[ "\text{LENGTH}(\text{SORT} \ L) = \text{LENGTH} \ L" ] \\
() : \text{void}
\end{align*}
\]

leaving ourselves with an easy auxiliary lemma to prove. We set a fresh goal:

\[
\begin{align*}
\text{#g} \ "(L:\text{num list}) \ a. \ \text{LENGTH}(\text{insert} \ a \ L) = \text{SUC}(\text{LENGTH} \ L)" ; ; \\
"(L. \ a. \ \text{LENGTH}(\text{insert} \ a \ L) = \text{SUC}(\text{LENGTH} \ L)" \\
() : \text{void} \\
\text{#e} (\text{LIST\_INDUCT\_TAC\ \text{THEN\ \text{REPEAT\ \text{GEN\_TAC}}}} \\
\text{\hspace{1cm} \text{THEN\ \text{REWITE\_TAC [ \text{LENGTH}; \text{insert} ]}}) ; ; \\
\text{OK} . . \\
"\text{LENGTH}(a <= h \Rightarrow \text{CONS} \ a(\text{CONS} \ h \ L) \ | \ \text{CONS} \ h(\text{insert} \ a \ L)) = \text{SUC}(\text{SUC}(\text{LENGTH} \ L))"
[ "a. \ \text{LENGTH}(\text{insert} \ a \ L) = \text{SUC}(\text{LENGTH} \ L)" ] \\
() : \text{void}
\end{align*}
\]

and bool-case on \( a <= h \).

\[
\begin{align*}
\text{#e} (\text{BOOL\_CASES\_TAC \ "a<=h"}} \\
\text{\hspace{1cm} \text{THEN\ \text{ASM\_REWRITE\_TAC [ \text{LENGTH} ]}) ; ; \\
\text{OK} . . \\
\text{goal\ proved} \\
. \vdash \text{LENGTH}(a <= h \Rightarrow \text{CONS} \ a(\text{CONS} \ h \ L) \ | \ \text{CONS} \ h(\text{insert} \ a \ L)) = \text{SUC}(\text{SUC}(\text{LENGTH} \ L)) \\
\vdash !L. \ a. \ \text{LENGTH}(\text{insert} \ a \ L) = \text{SUC}(\text{LENGTH} \ L) \\
\text{Previous\ subproof:} \\
\text{goal\ proved} \\
() : \text{void}
\end{align*}
\]
Here are the tidied lemmata.

```haskell
#let lem6 = prove
("! (L:num list) a . LENGTH(insert a L) = SUC(LENGTH L)",
  LIST_INDUCT_TAC THEN REPEAT GEM_TAC
  THEN REWRITE_TAC [LENGTH; insert]
  THEN BOOL_CASES_TAC "a<=[L"
  THEN ASM_REWRITE_TAC [LENGTH;; lem6;; LENGTH ;;];;
lem6 = |- (! L a. LENGTH(insert a L) = SUC(LENGTH L)
Run time: 0.5s
Intermediate theorems generated: 142

#let lem7 = prove
("! (L:num list). LENGTH(SORT L) = LENGTH L",
  LIST_INDUCT_TAC THEN REPEAT GEM_TAC
  THEN ASM_REWRITE_TAC [SORT; lem6; LENGTH ;;];;
lem7 = |- (! L. LENGTH(SORT L) = LENGTH L
Run time: 0.4s
Intermediate theorems generated: 98
```

(iv) \(\vdash ! L x. \text{occs} x \text{(SORT} L) = \text{occs} x L\)

The final lemma is the most interesting. We first remind you of the definition of \text{occs} and then start the proof.
- which suggests we prove the auxiliary lemma below. We set another goal and continue:

```lisp
# occs ;;
| ! (x. occs x[] = 0) /\ 
  | (!x a L. occs x(CONS a L) = ((x = a) => SUC(occs x L) | occs x L))
#g "! L x . occs x (SORT L) = occs x L";;
"!L x. occs x(SORT L) = occs x L"
(): void

#e(LIST_INDUCT_TAC THEN REPEAT GEN_TAC 
  THEN REWRITE_TAC [ SORT; occs ]);;;
OK .
"occs x(insert h(SORT L)) = ((x = h) => SUC(occs x L) | occs x L)"
[ "!x. occs x(SORT L) = occs x L" ]
(): void

#e(ASM_CASES_TAC "(x:num)<h"
  THEN ASM_REWRITE_TAC [] ];;;
OK .
2 subgoals
"occs x(insert h(SORT L)) = occs x L"
[ "!x. occs x(SORT L) = occs x L" ]
[ "(x = h)" ]
"occs h(insert h(SORT L)) = SUC(occs h L)"
[ "!x. occs x(SORT L) = occs x L" ]
[ "h = h" ]
(): void
```

```lisp
#g "! L x a .
  occs x (insert a L)
    = ((x=a) => SUC(occs x L) | occs x L)";;
"!L x a. occs x(insert a L) = ((x = a) => SUC(occs x L) | occs x L)"
(): void

#e(LIST_INDUCT_TAC THEN REPEAT GEN_TAC 
  THEN REWRITE_TAC [ insert; occs ]);;;
OK .
"occs x(a <= h => CONS a(CONS h L) | CONS h(insert a L)) = 
  ((x=a) => SUC((x = h) => SUC(occs x L) | occs x L) | 
  (x = h) => SUC(occs x L) | occs x L))"
[ "! a. occs x(insert a L) = ((x = a) => SUC(occs x L) | occs x L)" ]
(): void
```
We now bool-case on $a \leq h$ to simplify the if-then-else structure on the left and $x = h$ to simplify the if-then-else structure on the right.

Here we could bool-case yet again on $h = a$, pull the SUC inside the if-then-else structure or, as below, recast the assumption $\neg a \leq h$ into a directly useful form.
Here are the last two lemmata in tidy form.

```plaintext
#let lem8 = prove
("! L x a .
  occs x (insert a L)
  = (x=a) => SUCC (occs x L) | occs x L)",
LIST_INDUCT_TAC THEN REPEAT GEN_TAC
THEN REWRITE_TAC [ insert; occs ]
THEN MAP_EVERY ASM_CASES_TAC [ "(x:num) = h"; "a <= h" ]
THEN ASM_REWRITE_TAC [ occs ]
THEN IMP_RES_TAC trivs2
THEN ASM_REWRITE_TAC ( ) ; ; ;
lem8 =
|- !L x a . occs x (insert a L) = ((x = a) => SUCC (occs x L) | occs x L)
Run time: 2.9s
Garbage collection time: 0.7s
Intermediate theorems generated: 520
```
Our last step is to join the four principle lemmata into one main theorem for saving.

```
#let lem9 = prove
("! L . occs x (SORT L) = occs x L", 
 LIST_INDUCT_TAC THEN REPEAT GEN_TAC
 THEN REWRITE_TAC [ SORT; occs ]
 THEN PURE_ONCE_REWRITE_TAC [ lem8 ]
 THEN BOOL_CASES_TAC "(x:num)=h"
 THEN ASM_REWRITE_TAC []);

lem9 = |- !L x. occs x(SORT L) = occs x L
Run time: 0.6s
Intermediate theorems generated: 126

#let insertSortThms = save_thm
('insertSortThms',
 CONJ lem3 (CONJ lem5 (CONJ lem7 lem9)));
insertSortThms = 
|- (!L. sorted(SORT L)) /
 (!!L. isMem x(SORT L) = isMem x L) /
 (!!L. LENGTH(SORT L) = LENGTH L) /
 (!!L. occs x(SORT L) = occs x L)
Run time: 0.1s
Intermediate theorems generated: 3
```
Chapter 20

pico compiler correctness

20.1 Proving the correctness of a compiler

An introduction should be provided — read [37, 94].

20.2 Formal verification of pico

After logging in

```ml
#new theory 'pico';;
() : void

#let axiomatiseDataType stem defn
  = let axiom = define_type stem defn
    in
    let induct = prove_induction_thm axiom
    and one_one = prove_constructors_one_one axiom
    and distinct = (prove_constructors_distinct axiom) \ $\top$
    in
    let cases = prove_cases_thm induct
    and induct_tac = INDUCT_THEN induct ASSUME_TAC
    in
      (axiom, induct, one_one, distinct, cases, induct_tac);
axiomatiseDataType =
  :
    (string -> string -> (thm # thm # thm # thm # thm # tactic)))
```

we define the data type for expressions
We next define the machine instructions
and the translation schema from pico down to STC machine code

Finally we define some auxiliary functions chosen to make the definition of `exec` look crisp.
#let doAND = new_definition
('doAND',
"doAND (ST : bool list)
   = CONS ((HD (TL ST)) \ (HD ST) (TL (TL ST)))\n\);
doAND = |- ! ST. doAND ST = CONS (HD (TL ST) \ HD ST) (TL (TL ST))\n#let doOR = new_definition
('doOR',
"doOR (ST : bool list)
   = CONS ((HD (TL ST)) \ (HD ST) (TL (TL ST)))\n\);
doOR = |- ! ST. doOR ST = CONS (HD (TL ST) \ HD ST) (TL (TL ST))\n
#let BVAL = new_recursive_definition
false
STC_Axiom
'BVAL:'
"BVAL (STC_load b) = b\nBVAL = |- ! b. BVAL(STC_load b) = b\n
#let EXEC = new_recursive_definition
false
list_Axiom
'EXEC:'
"(EXEC (ST : bool list) ([] : STC list) = HD ST) \n(EXEC ST (CONS instr cont) = (? b. instr = (STC_load b)) \n  \ EXEC (CONS (BVAL instr) ST) cont 
  | (instr = STC_not) \ EXEC (doNEG ST) cont 
  | (instr = STC_and) \ EXEC (doAND ST) cont 
  | (instr = STC_or) \ EXEC (doOR ST) cont 
  | HD ST 
)\nEXEC = . . .\n
The load instruction, ? b . instr = STC_load b is the only one to present any difficulty. The scope of b is instr = STC_load b, but we require the value of b in the continuation EXEC (CONS b ST) cont. The BVAL abstraction does the trick (not that this would generalise to any other instructions with arguments).

We get rid of one trivial lemma before embarking on our proofs.

#let lem1 = prove
("! (b : bool). (? b'. b = b')",
GEN_TAC
THEN EXISTS_TAC "b : bool"
THEN REPL_TAC);;
lem1 = |- ! b. ? b'. b = b'
(i) "ST CODE. EXEC ST(APPEND(TRANSLATE P) CODE) = EXEC(CONS(EVAL P ST) CODE)"

After setting the goal, we induct over all program phrases, strip away any generalisations, and examine the cases one by one.

CON b

We take it step by step to show how the proof unfolds, first rewriting with TRANSLATE on the left and with EVAL on the right.
APPEND takes the argument to the singleton list STC_load b and CONSes it onto CODE

```haskell
{-#PURE_REWRITE_TAC [APPEND]#};;
OK.
"EXEC ST(CONS(STC_load b)CODE) = EXEC(CONS b ST)CODE"
() : void
```

and now we can rewrite with EXEC

```haskell
{-#PURE_REWRITE_TAC [EXEC]#};;
OK.
"((?b'. STC_load b = STC_load b') =>
 EXEC(CONS(BVAL(STC_load b))ST)CODE |
 ((STC_load b = STC_not) =>
 EXEC(doNEG ST)CODE |
 ((STC_load b = STC_end) =>
 EXEC(doAND ST)CODE |
 ((STC_load b = STC_or) => EXEC(doOR ST)CODE | HD ST))) =
 EXEC(CONS b ST)CODE"
() : void
```

In the next steps we simplify ((?b'. STC_load b = STC_load b') using STC_11

```haskell
#STC_11;
?- !b b'. (STC_load b = STC_load b') = (b = b')

{-#PURE_REWRITE_TAC [STC_11]#};;
OK.
"((?b'. b = b') =>
 EXEC(CONS(BVAL(STC_load b))ST)CODE |
 ((STC_load b = STC_not) =>
 EXEC(doNEG ST)CODE |
 ((STC_load b = STC_end) =>
 EXEC(doAND ST)CODE |
 ((STC_load b = STC_or) => EXEC(doOR ST)CODE | HD ST))) =
 EXEC(CONS b ST)CODE"
() : void
```

and then use STC_DISTINCT to show that the three other instruction comparisons are all false
We are nearly there. A quick rewrite with \texttt{lem1}

\begin{verbatim}
#STC_DISTINCT;;
| ~(!b. ~(STC_load b = STC_not)) /
| (!b. ~(STC_load b = STC_and)) /
| (!b. ~(STC_load b = STC_or)) /
| (!b. ~(STC_load b = STC_hlt))
| ~(STC_not = STC_and) /
| ~(STC_not = STC_or) /
| ~(STC_not = STC_hlt) /
| ~(STC_and = STC_or) /
| ~(STC_and = STC_hlt) /
| ~(STC_or = STC_hlt)

#(PURE_REWRITE_TAC [ STC_DISTINCT ]);;
OK...
"((?b'. b = b') =>
 EXEC(CONS(BVAL(STC_load b))ST)CODE | 
 (F =>
 EXEC(doNEG ST)CODE | 
 (F =>
 EXEC(doAND ST)CODE | 
 (F =>
 EXEC(doOR ST)CODE | 
 (F =>
 EXEC(doHLD ST)CODE | 
 (F =>
 EXEC(CONS b ST)CODE)) = 
 EXEC(CONS b ST)CODE) = 
 EXEC(CONS b ST)CODE")

() : void
\end{verbatim}

followed by another with \texttt{BVAL} leaves us with a reflection.
NOT P

We first rewrite with \texttt{TRANSLATE} and \texttt{EVAL}

\begin{verbatim}
#e(PURK_ORCH_REWRITE_TAC [ EVAL ]);;
OK..
"EXEC(CONS b ST)CODE = EXEC(CONS b ST)CODE"

(); void
#e REFL_TAC;;
OK..
goal proved

\%
Previous subproof:
3 subgoals
"EXEC ST(APPEND (TRANSLATE (OR P P')) CODE) =
EXEC (CONS (EVAL (OR P P')) ST) CODE"
  [ "ST CODE.
    EXEC ST (APPEND (TRANSLATE P) CODE) = EXEC (CONS (EVAL P) ST) CODE" ]
  [ "ST CODE.
    EXEC ST (APPEND (TRANSLATE P') CODE) = EXEC (CONS (EVAL P') ST) CODE" ]

"EXEC ST (APPEND (TRANSLATE (AND P P')) CODE) =
EXEC (CONS (EVAL (AND P P')) ST) CODE"
  [ "ST CODE.
    EXEC ST (APPEND (TRANSLATE P) CODE) = EXEC (CONS (EVAL P) ST) CODE" ]
  [ "ST CODE.
    EXEC ST (APPEND (TRANSLATE P') CODE) = EXEC (CONS (EVAL P') ST) CODE" ]

"EXEC ST (APPEND (TRANSLATE (NOT P)) CODE) =
EXEC (CONS (EVAL (NOT P)) ST) CODE"
  [ "ST CODE.
    EXEC ST (APPEND (TRANSLATE P) CODE) = EXEC (CONS (EVAL P) ST) CODE" ]

(); void
\end{verbatim}

and then use the reversed version of \texttt{APPEND_ASSOC} to push the left hand side into the right shape for rewriting from the assumption list.
As before, we now rewrite with \texttt{APPEND} to introduce an explicit \texttt{CONS} operation and rewrite with \texttt{EXEC}.

Rewriting with \texttt{STC\_DISTINCT} doesn't do everything we would like
because although it contains all four cases for \texttt{STC\_not} only three of them are “the right way round” (two for \texttt{STC\_and}, one for \texttt{STC\_or} and none at all for \texttt{STC\_hlt}). We need to take the conjuncts of \texttt{STC\_DISTINCT} and turn them round. This we can do using \texttt{CONJ\_LIST}. \texttt{CONJ\_LIST n th} expects the theorem \texttt{th} to be (at least \texttt{n}) theorems conjoined together. It returns the first \texttt{n} in a list. Mapping \texttt{GSYM} down \texttt{CONJ\_LIST 10} \texttt{STC\_DISTINCT} gives us what we want.
After rewriting with doNEG

we obtain a reflection when we simplify the goal with standard list rewrites

% % % % trace omitted % % % %
Cases AND \( e_1 \) \( e_2 \) and OR \( e_1 \) \( e_2 \)

The last two legs of the proof are very much the same as that of \( \text{\textsc{not}} \ b \) and teach us nothing new.

The proof in tidy form

\[
\text{let correct1} = \text{prove_thm}
\]

\[
('\text{correct1}',
\)

\[
'\text{P ST Code}'
\]

\[
\text{exec ST (append (translate P) CODE)}
\]

\[
= \text{exec (cons (eval P) ST) CODE''},
\]

\[
\text{Expr.induct_tac then repeat gen_tac}
\]

\[
\text{then pure.once_rewrite_tac [ translate; eval ]}
\]

\[
\text{then}
\]

\[
\text{let tac1} =
\]

\[
\text{map\_every pure\_rewrite\_tac [ [append]; [exec] ]}
\]

\[
\text{then rewrite\_tac [ stc\_ii; lem; bval; stc\_distinct ]}
\]

\[
\text{and tac2 thm} =
\]

\[
\text{asm\_rewrite\_tac [ gsym append\_assoc ]}
\]

\[
\text{then map\_every pure\_once\_rewrite\_tac [ [append]; [exec] ]}
\]

\[
\text{then rewrite\_tac [ stc\_distinct ]}
\]

\[
\text{then rewrite\_tac [ map gsym (comb\_list 10 stc\_distinct) ]}
\]

\[
\text{then pure.once\_rewrite\_tac [ thm ]}
\]

\[
\text{then rewrite\_tac [ hd; tl; append ]}
\]

\[
in
\]

\[
[ \text{tac1; tac2 doNeg; tac2 doAnd; tac2 doOr }];
\]

\[
\text{correct1} =
\]

\[
|! P \text{ ST Code} .
\]

\[
\text{exec ST(append(translate P)CODE)} = \text{exec(cons(eval P)ST)CODE}
\]

Run time: 5.5s

Garbage collection time: 0.7s

Intermediate theorems generated: 1553

(ii) \( 'P \cdot \text{exec [] (translate P)} = \text{eval P} \)

We set the goal, induct and perform some obvious rewrites
CON b

We strip away the generalisation and rewrite with EXEC

As in the previous theorem, we first simplify the conditions
Borrowing from the previous proof, we rewrite with \texttt{EXEC} and the theorems of \texttt{STC\_DISTINCT} reversed.

All that now remains is to rewrite with \texttt{doNEG}, \texttt{HD}, and \texttt{tl}.
And e1 e2 and Or e1 e2

Teach us nothing new.

Proof in tidy form

```plaintext
#let correct2 = prove_thm
('correct2',
  `! P . EXEC ([] : bool list) (TRANSLATE P) = EVAL P",
  Expr_INDUCT_TAC
  THEN PURE_REWRITE_TAC [ TRANSLATE; EVAL; correct1 ]
  THEN
  let tac thm =
    PURE_REWRITE_TAC [ EXEC ]
    THEN REWRITE_TAC (map GSYM (CONJ_LIST 10 STC_DISTINCT))
    THEN REWRITE_TAC [ thm; HD; TL ]
    in
    [ GEN_TAC
      THEN REWRITE_TAC [ EXEC ]
      THEN REWRITE_TAC [ STC_11; STC_DISTINCT ]
      THEN REWRITE_TAC [ lem1; BVAL; HD ]
      ;
      tac doNEG; tac doAND; tac doOR
    ]
  );
  correct2 = |- !P . EXEC [] (TRANSLATE P) = EVAL P
Run time: 3.9s
Garbage collection time: 0.7s
Intermediate theorems generated: 1075
```
EXERCISES 20

Exercise 20.1 Prove that

"\texttt{! P ST CODE . b . EXEC ST (APPEND (TRANSLATE P) CODE) = EXEC (CONS b ST) CODE}"?

```ocaml
#let correct3 = prove_thm
('correct3',
 " \texttt{! P ST CODE . ? b . EXEC ST (APPEND (TRANSLATE P) CODE) = EXEC (CONS b ST) CODE},
 REWRITE_TAC [ correct1 ]
 THEN Expr_INDUCT_TAC THEN REPEAT GEN_TAC
 THEN REWRITE_TAC [ EVAL ]
 THENL
 [ EXIST_TAC "b:bool" THEN REFL_TAC;
  EXIST_TAC " (EVAL:Expr\rightarrow bool) P" THEN REFL_TAC;
  EXIST_TAC " (EVAL:Expr\rightarrow bool) P" THEN REFL_TAC;
  EXIST_TAC " (EVAL:Expr\rightarrow bool) P" THEN REFL_TAC
 ]);
correct3 =
1- ! P ST CODE.
 ?b . EXEC ST (APPEND (TRANSLATE P) CODE) = EXEC (CONS b ST) CODE
```

Exercise 20.2 Prove that the evaluation of a well-formed expression increases the stack height by 1. This is a very useful theorem for a compiler writer. Similarly, in languages with commands, a evaluation of a well-formed command will leave the stack height unaltered.
Chapter 21

Verification of FSMs

Finite state machines are frequently used to describe hardware systems and subsystems [75]. As specifications, these descriptions are symbolic. No encodings are given for the states, the inputs, or the outputs. As implementations, Boolean representations are used entirely.

Given a fully symbolic finite-state machine specification, how do we know a Boolean-level description does “the same thing”? Conventional computer-aided design verification techniques rely on simulation [73]. Simulation, of course, can only cover a subset of all possible input sequences, and is thus only a partial solution. Also, given two implementations of the same specification, we should be able to show that they are “the same”, even though they might use very different Boolean encodings. Current CAD tools do this by comparing the state-transition behaviour at the Boolean level [93]. Nevertheless, this still does not directly relate a symbolic specification with its Boolean implementation.

In this chapter we will show how to relate symbolic specifications to their Boolean implementations by using the properties of data types. By so doing, we will be able to relate implementations of the same specification with ease.

21.1 Finite-state machines

Finite-state machines are described by a 5-tuple, \((S, I, O, \delta, \lambda)\) where

- \(S, I,\) and \(O\) are the state, input, and output alphabets respectively;
- \(\delta\) is the state transition function, \(S \times I \rightarrow S\); and
- \(\lambda\) is the output function: \(S \times I \rightarrow O\) for Mealy machines, and \(S \rightarrow O\) for Moore machines.

Two machines are said to be equivalent if the outputs from the two machines are identical for all input sequences when they start in corresponding initial states. We can extend this notion of equivalence to specification and implementation descriptions by incorporating functions which map states, outputs, and inputs from one description to another. This is illustrated
below (figure 20.1) where the pair \((S_1, I_{n_1})\) is the initial state and input at the specification level, \((M_1, M_{n_1})\) is the initial state and input pair at the implementation level, \(\text{state-trans}\) and \(\text{M-state-trans}\) are the state transformation functions of the specification and implementation respectively, and \(\text{REP\_state}, \text{REP\_input},\) and \(\text{ABS\_state}\) are the functions which give the mapping from the specification to the implementation and back again.

\[
\begin{align*}
\text{Specification : } & S_1, I_{n_1} \xrightarrow{\text{state-trans}} S_2 \\
& (\text{REP\_state, RE\_Pinput}) \uparrow \text{ABS\_state} \\
\text{Implementation : } & M_1, M_{n_1} \xrightarrow{\text{M-state-trans}} M_2
\end{align*}
\]

Figure 21.1 Relating Specification and Implementation Behaviours

A similar situation holds for generating the outputs (figure 20.2).

\[
\begin{align*}
\text{Specification : } & S_1, I_{n_1} \xrightarrow{\text{output\_table}} \text{out} \\
& (\text{REP\_state, RE}\_Pinput) \uparrow \text{ABS\_output} \\
\text{Implementation : } & M_1, M_{n_1} \xrightarrow{\text{M_output\_table}} \text{Mout}
\end{align*}
\]

Figure 21.2 Relating Specification and Implementation Output Functions

The above diagrams give us a “roadmap” for creating HOL theories which support finite-state machine verification and synthesis. From a verification perspective the diagrams indicate that to show equivalence between two machines, the various mapping functions between the two implementations must commute, e.g. going from \((S_1, I_{n_1})\) to \(S_2\) via state-trans is the same as starting from \((S_1, I_{n_1})\) and going through \((M_1, M_{n_1})\) and \(M_2\) for all states and inputs.

From a synthesis perspective, given the state-transition and output functions, once we select the state, input and output representations to be used in our implementations, the state-transition and output functions of the implementation should be fully determined. If these functions are at the Boolean level, then they can be used by conventional CAD tools to create gate-level implementations.
In the next section, we will develop parameterized functions and theorems which support these two perspectives. Later on, we will demonstrate the use of these functions and theorems by developing two different but equivalent-behaving implementations of a traffic light controller.

### 21.2 Evaluation functions and theorems

In this section we initially define functions which evaluate the state-transition and output functions and map them to equivalent-behaving state-transition and output functions given state, input, and output mapping functions. Afterwards, we devote the remainder of the section to proving properties about the defined functions.

#### 21.2.1 Evaluation functions

We first define a higher-order function, `foldl`, which takes as inputs 1) a binary operator `op`; 2) a single element `a`, usually the identity of operator `op`; and 3) a list of elements of the same type of `a`.

```haskell
#let foldl = new_recursive_definition
  false
  list_Axiom
  'foldl'
  
  "(foldl (op:(**)) -> *) (a:*):((**))list) = a)
  \/
  (foldl op a (CONS x xs) = foldl op (op(a,x)) xs)"

foldl = ...
```

When `a` is the identity element of `op`, then `foldl` corresponds to the `reduce` operation, e.g.

\[
\begin{align*}
\text{foldl} + 0 [1; 2; 3] &= \text{foldl} + (0+1) [2; 3] \\
&= \text{foldl} + (0+1+2) [3] \\
&= \text{foldl} + (0+1+2+3) [4] \\
&= 0+1+2+3
\end{align*}
\]

The state evaluation function, `eval_S`, is defined using `foldl`. `eval_S` returns a final state when given 1) a state-transition function, 2) a list of inputs, and 3) an initial state. The state-transition function is the binary operator supplied to `foldl`, `lst` is the list of inputs, and `s` is the initial state.
The next function we define, \( \text{eval}_O \), is used for collecting the outputs given a state transition function \( s_{\text{trans}} \), an output function \( o_{\text{table}} \), an initial state \( s \), and a list of inputs. Its function is diagrammed in Figure 21.3.

\[
\begin{array}{c}
\text{let eval}_S = \text{new definition} \\
\text{false} \\
\text{list Axiom} \\
\text{ `eval}_S` \\
\text{`eval}_S (s_{\text{trans}:(*\rightarrow *) \ (1\text{st}:(**)) \ (s:*)) = \text{fold1} \ s_{\text{trans} \ s \ \text{list})`}; \\
\text{eval}_S = ...
\end{array}
\]

\[
\begin{array}{c}
\text{let eval}_O = \text{new recursive definition} \\
\text{false} \\
\text{list Axiom} \\
\text{ `eval}_O` \\
\text{`eval}_O (s_{\text{trans}:(*\rightarrow *)) \\
(o_{\text{table}:(*\rightarrow **)) \ (s:*)) \\
(\text{[]}:(**))\text{list}) = \text{[]} /\ \\
\text{eval}_O s_{\text{trans}} o_{\text{table}} s \ (\text{CONS} \ i \ \text{rest}) = \text{CONS} \ (o_{\text{table}}(s,i)) \ (\text{eval}_O s_{\text{trans}} o_{\text{table}} (s_{\text{trans}}(s,i) \ \text{rest})); \\
\text{eval}_O = ...
\end{array}
\]

21.2.2 Parametric mapping functions

In this section, we define two parametric mapping functions which create equivalent-behaving state-transition and output functions given state, input, and output mapping functions. The first function, \( M_{\text{state,trans}} \), returns an equivalent state transition function given 1) a specified state-transition function, 2) a state encoding function, 3) a state representation decoding function, and 4) an input representation decoding function. Later on we will use this function to synthesize a Boolean state-transition function from a symbolic specification given particular Boolean state and input representations.
21.2. EVALUATION FUNCTIONS AND THEOREMS

\[
MS_1, M\text{In}_1 \xrightarrow{M_{\text{state\_trans}}} MS_2
\]

\[
\begin{align*}
S_1, I\text{In}_1 & \xrightarrow{\text{state\_trans}} S_2 \\
\end{align*}
\]

Figure 21.4 Defining the Machine behaviour

```
#let M_state_trans = new_definition
  ('M_state_trans',
   "M_state_trans
   (s_trans:*## -> *)
   (encode_s:* -> ***)
   (decode_s:*** -> *)
   (decode_i:(*4) -> **)
   (y:***,x:(**))
   = encode_s(s_trans(decode_s(y),decode_i(x)))
   ");
M_state_trans = ...
```

The second parametric mapping function we define is \texttt{M\_output\_table}. \texttt{M\_output\_table} generates an equivalent output function given 1) a specified output function, 2) an output encoding function, 3) a state representation decoding function, and 4) an input representation decoding function. Like the previous function, given particular Boolean state, input, and output representations, a symbolic output table will be synthesized to its Boolean equivalent.

\[
MS_1, M\text{In}_1 \xrightarrow{M_{\text{output\_table}}} M\text{out}
\]

\[
\begin{align*}
S_1, I\text{In}_1 & \xrightarrow{\text{output\_table}} \text{out} \\
\end{align*}
\]

Figure 21.5 Defining the machine output
21.2.3 Proving the correctness of the parametric mapping functions

Given the definitions in the previous sections, we will prove that the parametric mapping functions give us the desired commutative behaviour as illustrated in Figures 21.1 and 21.2.

The first theorem we prove states that given two state transition functions \texttt{s\_trans} and \texttt{M\_s\_trans}, \texttt{eval\_S} returns equivalent final states for each when 1) both are supplied corresponding states, 2) the input list of one corresponds to an encoding of the other’s input list, 3) the state encodings are invertible, and 4) given corresponding state and input pairs, the states returned by \texttt{s\_trans} and \texttt{M\_s\_trans} also correspond.
We prove the theorem by induction on lists.
Base case. The base case is proved by using the assumption that the state encoding is invertible, i.e. \( \forall s'. \ decode_s(encode_s s') = s' \) and from the definitions of \texttt{eval_i}, \texttt{MAP}, and \texttt{foldl}.

### Evaluation Functions and Theorems

**Induction step.** To prove the recursive case we make the following substitutions in the assumptions:

- \( s_{\text{trans}}(s, h) \) \( \Rightarrow s \)
- \( \text{encode}_s(s_{\text{trans}}(s, h)) \) \( \Rightarrow y \)
- \( \text{MAP encode}_i \text{ inL} \) \( \Rightarrow xl \)
- \( \text{encode}_s \) \( \Rightarrow \text{encode}_s \)
- \( \text{encode}_i \) \( \Rightarrow \text{encode}_i \)
- \( s_{\text{trans}} \) \( \Rightarrow s_{\text{trans}} \)

Additionally, we specialize the remaining variables to their quantified names and expand using the definition of \( \text{eval}_S \). This eliminates the first two conjuncts of the antecedent. We do these substitutions by mapping the composition of \texttt{REWRITE\_RULE}, \texttt{SPEC\_ALL}, and \texttt{SPECL} over the single assumption using \texttt{RULE\_ASSUM\_TAC}. 

```haskell
#MAP

| (f, MAP f[]) = [] ) /
| (f h t, MAP f(CONS h t) = CONS(f h)(MAP f t))

#e STRIP\_TAC THEN RES\_TAC

THEN ASM\_REWRITE_TAC [ eval_S; MAP; foldl ]);

OK;

goal proved

| (y = encode_s s) /
| (xL = MAP encode_i[]) /
| (!s', decode_s(encode_s s') = s') /
| (!s' x, encode_s(s_{\text{trans}}(s',x)) = M_{s_{\text{trans}}(encode_s s',encode_i x))} ||
| (eval_S s_{\text{trans}}[]) = decode_s(eval_S M_{s_{\text{trans}} xL y})

Previous subproof:

| (y = encode_s s) /
| (xL = MAP encode_i(CONS h inl)) /
| (!s', decode_s(encode_s s') = s') /
| (!s' x, encode_s(s_{\text{trans}}(s',x)) = M_{s_{\text{trans}}(encode_s s',encode_i x))} ||
| (eval_S s_{\text{trans}}(CONS h inl)) = decode_s(eval_S M_{s_{\text{trans}} xL y})" ||
| [ "y = encode_s s) /
| (xL = MAP encode_i inl) /
| (!s', decode_s(encode_s s') = s') /
| (!s' x, encode_s(s_{\text{trans}}(s',x)) = M_{s_{\text{trans}}(encode_s s',encode_i x))} ||
| (eval_S s_{\text{trans}} inl s = decode_s(eval_S M_{s_{\text{trans}} xL y})" ]

() : void
```
Next, we simplify the goal by using STRIP_TAC and generate additional assumptions by applying RES_TAC. All that is left then is a rewrite using the assumptions.
Here is the proof in tidy form
The next theorem we prove is similar to the `state_match_thm` except it pertains to generating lists of outputs. Basically, given the same assumptions as in `state_match_thm`, plus the assumption that we have corresponding output functions, the list of outputs generated given corresponding input lists will be the same.
We use list induction and simplify the goal by removing all universal quantifiers.
Base case. The base case is proved by moving all of the antecedents of the goal into the assumption list by using \texttt{STRIP_TAC}, enriching the assumptions by using \texttt{RES_TAC}, and by rewriting the goal using the assumptions and the definitions of \texttt{evalo} and \texttt{MAP}.
Induction step. We first remind you of the standard definitions:

\[
\begin{align*}
TL &= |- \text{ht. } TL(\text{CONS h t}) = t \\
HD &= |- \text{ht. } HD(\text{CONS h t}) = h \\
NULL &= |- \text{NULL[]} /\ (\text{ht. } \text{NULL(CONS h t)}) \\
APPEND &= \\
|\ - (1. \text{APPEND[]}1 = 1) /\ \\
|\ (11 12 h. \text{APPEND(CONS h 11)12 = CONS h (APPEND 11 12))}
\end{align*}
\]

The recursive case is proved by rewriting the assumption using the four conjuncts of the antecedent of the goal. We will call the four conjuncts th1, th2, th3 and th4. The first seven universally quantified variables in the assumption, are specialized to:

\[
\begin{align*}
xL &\Rightarrow MAP \text{ encode_i } \text{inL} \\
s &\Rightarrow s_{\text{trans}}(s, h) \\
y &\Rightarrow \text{encode_s}(s_{\text{trans}}(s, h)) \\
\text{encode_s} &\Rightarrow \text{encode_s} \\
\text{encode_i} &\Rightarrow \text{encode_i} \\
\text{encode_o} &\Rightarrow \text{encode_o} \\
s_{\text{trans}} &\Rightarrow s_{\text{trans}}
\end{align*}
\]
After this, the goal is rewritten using the definitions and some of the four conjuncts.

Here is the proof in tidy form.
21.2.  EVALUATION FUNCTIONS AND THEOREMS

#let outputs_match_thm = prove_thm
"(!inL:(*)list)
(xL:((*)list)
(s::*)
(y::*)
(encode_s:(*->**))
(encode_i:*->(*4))
(encode_o:(*5)->(*6))
(s_trans:***->**)
(M_s_trans:***#(*4)->****)
(o_table:***#(*5))
(M_o_table:***#(*4)->(*6))
(decode_s:**->*)
(decode_i:(*4)->*)
(decode_o:(*5)->(*6)).

(y = encode_s s) /
(xL = MAP encode_i inL) /
(!s x. M_s_trans(encode_o s, encode_i x)
  = encode_o(s(s_trans(s,x))) /
(!s x. o_table(s,x)
  = decode_o(M_o_table(encode_s s, encode_i x)))
)

==» ( (eval_0 s_trans o_table s inL): ((*)list)
  = MAP
    (decode_o:(*5)->(*5))
    (eval_0 M_s_trans M_o_table y xL)
)

",
INDUCT_THEN list_INDUCT_ASSUME_TAC
THEN REPEAT GEN_TAC
THEN
[ STRIP_TAC THEN RES_TAC
  THEN ASM_REWRITE_TAC [ eval_0; MAP ]
];
DISCH_THEN (\th.
  let [ th1; th2; th3; th4] = CONJUNCTS th in
  \RULE_ASSUME_TAC
  ( ( REWRITE_RULE [ th1; th2; th3; th4; eval_0; APPEND; MAP ]
    o SPEC_ALL
    o ( SPEC
      \[ "MAP (encode_i:*->(*4)) inL";
      "(s_trans:***->**)(s,h)";
      "(encode_s:**->**)((s_trans:***->**))(s,h)";
      "encode_s:*->***";
      "encode_i:*->(*4)";
      "encode_o:(*5)->(*6)";
      "s_trans:***->**"\]
    )
  )
  THEN REWRITE_TAC [ th1; th2; eval_0; MAP ]
  THEN REWRITE_TAC [ NULL; MAP; HD; TL]
  THEN ASM_REWRITE_TAC [ th3; th4; eval_0 ]
)
)
);
outputs_match_thm = ...
The previous theorems on states and output lists dealt with arbitrary state-transition and output functions $M_{\text{trans}}$ and $M_{\text{output}}$. The next two theorems show the correctness of the functions when generated by $M_{\text{state}}$ and $M_{\text{output}}$.

```plaintext
#let M_state_trans_correct = prove_thm
  ('M_state_trans_correct',
   "!(encode_s::***->****)
   (decode_s::***->**)
   (encode_i::**->(*4))(decode_i::(*4)->**)
   (s_trans::**->**).
   (((s.decode_s(encode_s s)) = s) \/
   (i.decode_i(encode_i i)) = i)
  ) ==> (!s x. encode_s(s_trans(s,x))
   = (M_state_trans s_trans encode_s decode_s decode_i)
   (encode_s(s),encode_i(x)))")

","
REPEAT GEN_TAC
THEN REWRITE_TAC [M_state_trans]
THEN DISCH_THEN (\th. REWRITE_TAC (CONJUNCTS th)));
M_state_trans_correct = ...;

The proof of correctness for the output function is virtually the same:

```plaintext
#let M_output_table_correct = prove_thm
  ('M_output_table_correct',
   "!(encode_s::***->****)
   (decode_s::***->**)
   (encode_i::**->(*4))
   (decode_i::(*4)->**)
   (encode_o::(*4)->(*5))
   (decode_o::(*5)->(*6))
   (o_table::**->**).
   (((s.decode_s(encode_s s)) = s) \/
   (i.decode_i(encode_i i)) = i) \/
   (!out.decode_o(encode_o out) = out)
  ) ==> (!s x. o_table(s,x)
   = decode_o
   ((M_output_table o_table encode_o decode_o decode_i)
   (encode_s(s),encode_i(x))))")

","
REPEAT GEN_TAC
THEN REWRITE_TAC [M_output_table]
THEN DISCH_THEN (\th. REWRITE_TAC (CONJUNCTS th)));
M_output_table_correct = ...;

We now continue on to prove that the outputs generated by $M_{\text{output}}$ correspond to the outputs generated by output_table. First, we create a simple lemma under the assumption of the correctness of $M_{\text{state}}$ and $M_{\text{output}}$.

```
#let lemma_2 = REMOTE_RULE []
(SPECL [
  "inL:(*)list";
  "map (encode_i:*->(*)((inL:*))list)";
  "s:*";
  "((encode_s:*->*)s:*);";
  "encode_s:*->*";
  "decode_i:(*)*";
  "((encode_o:(*)->(*))s:*);";
  "s_trans:*#->*";
  "M_state_trans"s_trans:*#*->* (encode_s:*->*) (decode_s:*->*) (decode_i:(*)*);"
  "o_table:*#*->(*)";
  "M_output_table"o_table:*#*->(*) (encode_o:(*)*->*) (decode_s:*->*) (decode_i:(*)*);"
] outputs_match_thm);

lemma_2 = |- (!s' x.
  M_state_trans s_trans encode_s decode_s decode_i
  (encode_s s',encode_i x) =
  encode_s(s_trans(s',x))) /
  (!s' x.
   o_table(s',x) =
   decode_o (M_output_table o_table encode_o decode_s decode_i
   (encode_s s',encode_i x)))) =>
  (eval_0 s_trans o_table s inL =
   map decode_o (eval_0 (M_state_trans s_trans encode_s decode_s decode_i)
   (M_output_table o_table encode_o decode_s decode_i)
   (encode_s s)
   (map encode_i inL)))
We can get the final theorem showing the correspondence between the outputs as follows. The correspondence between the output lists is shown below in Figure 21.6.

\[ s \pi [i_1; i_2; \ldots; i_n] \]
\[ \text{(encode)} \downarrow \text{MAP encode} \]
\[ MS_1, [M_{n_1}; \ldots M_{in}] \xrightarrow{M_{trans}} S_2, [M_{i_2}; \ldots M_{in}] \xrightarrow{M_{trans}} \ldots \xrightarrow{M_{trans}} S_n, [M_{in}] \]
\[ M_{trans} \downarrow \]
\[ \begin{array}{c}
[M_{out_1};] \\
[M_{out_2};] \\
[\ldots] \\
[M_{out_n};]
\end{array} \]
\[ \text{decode} \downarrow \]
\[ \begin{array}{c}
[\text{out}_1;] \\
[\text{out}_2;] \\
[\ldots] \\
[\text{out}_n;]
\end{array} \]

Figure 21.6 Correspondence of outputs

\begin{verbatim}
#let M_outputs_match_thm = prove_thm
  ("M_outputs_match_thm",)
  "! s
    in
    (decode_s: *** \rightarrow **) \
    (encode_s: ** \rightarrow ***) \
    (decode_i: (*4) \rightarrow *) \
    (encode_i: * \rightarrow (*4)) \
    (decode_o: (*6) \rightarrow (*5))) \
    (encode_o: (*5) \rightarrow (*6))
  s_trans o_table .
  ( (!s. decode_s(encode_s s) = s) \land
    (!i. decode_i(encode_i i) = i) \land
    (!out. decode_o(encode_o out) = out))
) \Rightarrow
  (eval_0 s_trans o_table s inL
    = MAP decode_o
      (eval_0
        (M_state_trans s_trans encode_s decode_s decode_i)
        (M_output_table o_table encode_o decode_o decode_i)
        (encode_s s)
        (MAP encode_i inL))
  )

\end{verbatim}

\[ ^{1} M_{trans} \text{ in the figure represents } [M_{state} \downarrow s_{trans} \text{encode}_s \text{decode}_s \text{decode}_i] \] and \[ M_{table} \text{ represents } [M_{output} \downarrow \text{table encode}_o \text{decode}_o \text{decode}_i]. \]
21.3 Describing state machines using types

The functions and theorems developed for finite-state machines in the previous section were parameterized in terms of the mapping function from one machine description to the other. We saw that equivalent behaviour required that the mapping functions from the specification to the implementation had to be invertible.

In the previous two chapters we introduced techniques to create new data types and reason about their properties. We take advantage of these techniques to simplify the verification of equivalence of two finite state machine descriptions. Generally, the steps we will follow are these:

1. At the specification level where states, inputs, and outputs are symbolic, we will define them as enumerated types using define_type.

2. At the implementation level, states, inputs, and outputs will be defined in terms of existing types - often in terms of Boolean n-tuples, using new_type_definition.

3. Mapping functions between the states, inputs, and outputs of the specification and implementation will be defined using the type representation and abstraction functions.

4. The invertibility of the mapping functions will be proved which will imply the equivalence of the specification and implementation state machine descriptions.
We will also use types to reduce the number of cases we need to consider when dealing with structured data. For example, if the output of a simple controller at the intersection of two roads sets the colour the traffic lights for each road, we could describe the output as a pair \((c_1, c_2)\) where \(c_1\) and \(c_2\) specify the colour of each light. The possible colours are \text{green}, \text{yellow},\text{ and red.} There are of course nine possible pairs of colours including \((\text{green, green})\) which we would not include as a possible output. In fact, there are only four combinations we would want: \((\text{green, red}), (\text{yellow, red}), (\text{red, green}),\text{ and (red, yellow).}\) The other combinations would not be valid outputs. To capture this notion, we could define a new data type called output consisting of output symbols \(\text{GR}, \text{YR, RG,}\text{ and RY.}\) These symbols would be implemented in terms of the four valid pairs of colours. Thus, when reasoning about all outputs, we would be able to restrict our attention to the four pairs of valid colour combinations.

We will develop in detail the traffic light controller specification and two implementations. While there will be many definitions and theorems, the definitions and proofs will be very similar. To emphasize the common structure and simplify the presentation, we will follow the structure as shown in the table below for defining and proving properties of the states and colours for the controller. A \(\text{D}\) under role indicates a definition or property supporting the definitions of a data type. A \(\text{P}\) indicates a key property proved about a data type that is useful when using the data type. The definitions and properties of the controller specification's state and colour are made and proved using the automatic type-definition package in HOL.

<table>
<thead>
<tr>
<th>axioms</th>
<th>role</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{Z}^\text{axiom})</td>
<td>D</td>
</tr>
<tr>
<td>(\text{Z}^\text{induct})</td>
<td>P</td>
</tr>
<tr>
<td>(\text{Z}^\text{cases})</td>
<td>P</td>
</tr>
<tr>
<td>(\text{Z}^\text{distinct})</td>
<td>P</td>
</tr>
<tr>
<td>(\text{Z}^\text{EQ_Cases})</td>
<td>P</td>
</tr>
</tbody>
</table>

where \(Z = \text{state, Colour}\)

When constructing or reasoning about particular representations of data types, as is necessary for particular implementations, we cannot use the fully automatic data type package. Instead, we must establish the subset of the existing type to be used to represent the new data type, prove that the subset is not empty, establish the necessary representation and abstraction functions, and specify the particular mapping to be used. The development-
tal structure we use is shown in the following table. It is used for developing
the output data type, the binary encoding of the light colours, and the state
and output encoding for the two implementations.

<table>
<thead>
<tr>
<th>Function name</th>
<th>role</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS_Z</td>
<td>D</td>
</tr>
<tr>
<td>Z_exists</td>
<td>D</td>
</tr>
<tr>
<td>Z_TY_DEF</td>
<td>D</td>
</tr>
<tr>
<td>REP/ABS</td>
<td>D</td>
</tr>
<tr>
<td>decode_Z</td>
<td>D</td>
</tr>
<tr>
<td>IS_REP_Z</td>
<td>P</td>
</tr>
<tr>
<td>decode_REP_Z</td>
<td>P</td>
</tr>
<tr>
<td>Z_cases</td>
<td>P</td>
</tr>
<tr>
<td>Z_induct</td>
<td>P</td>
</tr>
<tr>
<td>REP_decode_Z</td>
<td>P</td>
</tr>
<tr>
<td>Z_distinct</td>
<td>P</td>
</tr>
<tr>
<td>Z_EQ_CLAUSES</td>
<td>P</td>
</tr>
<tr>
<td>where Z = Mcolour, output, M1state, M1output, M2state</td>
<td></td>
</tr>
</tbody>
</table>

In this section we present detailed examples of developing specification
and implementation descriptions for a finite-state machine used to control
the traffic lights at an intersection of a highway and a farmroad. This is
the same example originally presented in [75] and developed in VHDL by
[73]. Interestingly enough, we use a similar approach to doing the design
as [73], namely creating a high-level specification, defining representations
in terms of types built from Boolean structures, and deriving Boolean-
level descriptions of behaviour. Unlike [73], because we have all of HOL
and the logic behind data types, we will be able to formally prove our
implementation is equivalent to our specification. This eliminates the need
for creating test descriptions used by Lipsett to demonstrate that the VHDL
specification and implementation descriptions are the same.

21.3.1 Specification of the traffic light controller

Informal description

The traffic light controller we specify is used to control the intersection of
a highway and a farmroad. There are sensors on the farmroad to detect the
presence of cars. An input \texttt{Cars} to the controller is used to indicate the
presence of cars on the farmroad. Since the highway carries much more
traffic than the farm road, its light should change to red only when a car is detected on the farm road. Otherwise, it should remain green.

The highway light must remain green for some minimum amount of time. This amount of time is tracked by a timer which gives a signal $\text{Timeout}_L$. When $\text{Timeout}_L$ is true, then the minimum amount of time for which the highway light must be green has passed.

The yellow light must be on for some shorter period of time. This time is also tracked by a timer which gives a signal $\text{Timeout}_S$. When it is true, then the necessary period of time has passed.

For simplicity, we assume that a single timer produces both timer signals. We can start this timer by setting an output $\text{ST}$ to true. The other outputs, $\text{HL}$, and $\text{FL}$ indicate the colour of the highway and farm road lights, respectively.

The transition table describing the operation of the controller follows.

<table>
<thead>
<tr>
<th>This state</th>
<th>If inputs* are</th>
<th>Next state</th>
<th>Outputs</th>
<th>ST</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>HL</td>
<td>FL</td>
</tr>
<tr>
<td>Hway Green</td>
<td>(Cars) $\land$ ($\text{Timeout}_L$) = F</td>
<td>Hway Green</td>
<td>Green</td>
<td>Red</td>
</tr>
<tr>
<td></td>
<td>(Cars) $\land$ ($\text{Timeout}_L$) = T</td>
<td>Hway Yellow</td>
<td>Green</td>
<td>Red</td>
</tr>
<tr>
<td>Hway Yellow</td>
<td>$\text{Timeout}_S$ = F</td>
<td>Hway Yellow</td>
<td>Yellow</td>
<td>Red</td>
</tr>
<tr>
<td></td>
<td>$\text{Timeout}_S$ = T</td>
<td>Freoad Green</td>
<td>Yellow</td>
<td>Red</td>
</tr>
<tr>
<td>Freoad Green</td>
<td>(Cars) $\lor$ ($\text{Timeout}_L$) = F</td>
<td>Freoad Green</td>
<td>Red</td>
<td>Green</td>
</tr>
<tr>
<td></td>
<td>(Cars) $\lor$ ($\text{Timeout}_L$) = T</td>
<td>Freoad Yellow</td>
<td>Red</td>
<td>Green</td>
</tr>
<tr>
<td>Freoad Yellow</td>
<td>$\text{Timeout}_S$ = F</td>
<td>Freoad Yellow</td>
<td>Red</td>
<td>Yellow</td>
</tr>
<tr>
<td></td>
<td>$\text{Timeout}_S$ = T</td>
<td>Hway Green</td>
<td>Red</td>
<td>Yellow</td>
</tr>
</tbody>
</table>

* Inputs not listed are $\text{don't care}$. " \(^{-}\)" indicates inverse

### 21.3.2 HOL specification

**State data type using the automated type package**

We first introduce the state alphabet by defining a new data type called state. A theorem is returned which fully characterizes the new data type. Note, $?!$ denotes unique existence.

```plaintext
#let state_axiom = define_type
  'state_axiom'
  'state = HG | HY | FG | FY';
state_axiom =
  |- !e0 e1 e2 e3.
  ?! fn. (fn HG = e0) \land (fn HY = e1) \land (fn FG = e2) \land (fn FY = e3)
```

The properties of state like cases, induction, and distinctness are proved.
For later convenience, we also introduce the symmetric versions of each conjunct in the `states_distinct` theorem.

**Colour data type using the automated type package**

In the same way as above, we introduce the colours `red`, `yellow`, and `green` as a new data type called `colour`, and prove the same properties about `colour` as we did with `state`.
CHAPTER 21. VERIFICATION OF FSMS

Output definitions using existing types

The output of the controller is a 3-tuple \( \text{colour} \times \text{colour} \times \text{bool} \) where the first element is the colour of the Highway light, the second element is the colour of the farm road light, and the third element indicates if the timer should be started. The total number of distinct pairs is 18. However, only eight 3-tuples are valid outputs. To focus our attention only on the valid tuples, we create a new \text{output} data type from the eight valid 3-tuples. The output data type will be the set \{GRF, GRT, YRF, YRT, RGF, RGT, RYF, RYT\}. GRF will be represented by \((\text{Green}, \text{Red}, \text{F})\), GRT will be represented by \((\text{Green}, \text{Red}, \text{T})\), etc.

Since we have a concrete representation in mind for the data type, we must use a different set of functions that those used previously for defining states and colours. Specifically, we must

1. Define a subset of the representing type, (in this case a subset of the type \( \text{colour} \times \text{colour} \times \text{bool} \)). This is done by defining a predicate
on the representing type which identifies which elements are used and which are not.

2. Prove that the subset defined above is not empty.

3. Define the type using the predicate and the property that the underlying subset is non-empty.

4. Create the representation and abstraction functions used to map elements of the data type to their underlying representation.

5. Prove properties of the data type like
   - Any valid representation of an element of the data type can be mapped back to the original element.
   - Listing all cases of the data type.
   - Induction over the data type.
   - Any instance of the data type can be represented.
   - All instances of the data type are distinct from one another.
   - No element of the data type equals any other element.

We first define the predicate \texttt{IS\_output} which identifies which 3-tuples are valid output representations.

```plaintext
#let IS_output = new_definition
('IS_output',
"!out:colour#colour#bool.
IS_output out
  = (out = (Red, Yellow, T)) \/
   (out = (Red, Yellow, F)) \/
   (out = (Red, Green, T)) \/
   (out = (Red, Green, F)) \/
   (out = (Yellow, Red, T)) \/
   (out = (Yellow, Red, F)) \/
   (out = (Green, Red, T)) \/
   (out = (Green, Red, F))");
```

Next, we must show that this subset is not empty, i.e. that at least one 3-tuple is in the set defining the new data type. The proof is very easy. Basically, we show that the tuple \texttt{(Green, Red, F)} is valid. This can be done by \texttt{EXISTS_TAC}. We complete the proof by showing that \texttt{(Green, Red, F)} is not equal to any of the invalid combinations. This is done by rewriting using DeMorgan’s law, \texttt{PAIR_EQ}, and \texttt{colour_EQ_CLAUSES}. 

We introduce the type definition for the output data type by using `new_type_definition` and the `IS_output` definition with the `output_EXISTS` theorem.

The one-to-one and onto properties of the abstraction and representation functions are proved automatically. Notice that the invertibility of the representation and abstraction functions is proved automatically.
So far, we have not yet given specific names to each element in the new data type. The above definitions are just constants and the theorems specify their properties. We define another function, \texttt{decode_output}, which enables us to associate an element of the new data type with each valid 3-tuple. We define \texttt{decode_output} in terms of the abstraction function \texttt{ABS_output}.

Notice that \texttt{decode_output} also defines the mapping for all the don't care or invalid 3-tuples. In this case, we arbitrarily map all the don't care combinations to \((\text{Green}, \text{Red}, F)\). More complicated don't care mappings are possible. \texttt{decode_output} allows us to define the elements making up the output type in terms of their representation.

Using \texttt{decode_output}, we make the definitions for the output data type consisting of \texttt{GRF}, \texttt{GRT}, \texttt{YRF}, \texttt{YRT}, \texttt{RGF}, \texttt{RGT}, \texttt{RYF}, and \texttt{RYT}. Of course, a different valid representation must be associated with each symbol otherwise we will be unable to prove that we have a distinct representation.
We now prove a theorem which states that for any output \( a \), its representation is a valid one, i.e. \( \exists a. \text{IS}_\text{output}(\text{REP}_\text{output} \ a) \).

The next theorem we prove is \text{decode}_\text{output} inverts \text{REP}_\text{output} for all outputs.
21.3. DESCRIBING STATE MACHINES USING TYPES

We now proceed to prove the cases theorem for the outputs.

```
#let decode_REP_output = prove_thm
  ('decode_REP_output',
   "!out. decode_output(REP_output out) = out",
   GEN_TAC
   THEN REWRITE_TAC
   [ [ decode_output; IS_REP_output; REP_output_INVERTS ]];
decode_REP_output = |- !out. decode_output(REP_output out) = out
```

First, we remove the universal quantifier and expand the definition of each output term.

```
#g "!out. (out = GRF) \/
 (out = GRT) \/
 (out = YRF) \/
 (out = YRT) \/
 (out = RGF) \/
 (out = RGT) \/
 (out = RYF) \/
 (out = RYT)"

() : void
```

Next, we recall

```
ABS_output_DNOT = |- !a. ?r. (a = ABS_output r) / IS_output r
```
We use the first conjunct to substitute $\text{ABS}_\text{output} \ r$ for $\text{out}$ and the definition of $\text{IS}_\text{output}$ to generate the eight cases of interest.

```plaintext
#(CHOOSE_THEN
   (CONJUNCTS_THEN2
      SUBST1_TAC
      (\th. STRIP_ASSUME_TAC (REWRITE_RULE [ IS_output ] th)))
   (SPEC "out:output" ABS_output)(!))

OK..
8 subgoals

% << **** 7 subgoals omitted **** >> %

"(ABS_output r = decode_output(Gr,Red,F)) \\
(ABS_output r = decode_output(Gr,Red,T)) \\
(ABS_output r = decode_output(Yel,Red,F)) \\
(ABS_output r = decode_output(Yel,Red,T)) \\
(ABS_output r = decode_output(Re,Gr,F)) \\
(ABS_output r = decode_output(Re,Gr,T)) \\
(ABS_output r = decode_output(Re,Yel,F)) \\
(ABS_output r = decode_output(Re,Yel,T))"

[ "r = Gr,Red,T" ]

(): void

We finish off the proof by rewriting using the assumptions and the definition of $\text{decode}_\text{output}$.

```plaintext
#(ASM_REWRITE_TAC [ decode_output; IS_output ]);

OK..
goal proved

. 1- (ABS_output r = decode_output(Gr,Red,F)) \\
   (ABS_output r = decode_output(Gr,Red,T)) \\
   (ABS_output r = decode_output(Yel,Red,F)) \\
   (ABS_output r = decode_output(Yel,Red,T)) \\
   (ABS_output r = decode_output(Re,Gr,F)) \\
   (ABS_output r = decode_output(Re,Gr,T)) \\
   (ABS_output r = decode_output(Re,Yel,F)) \\
   (ABS_output r = decode_output(Re,Yel,T))

Previous subproof:
7 subgoals

% << **** rest of trace omitted **** >> %
```

The rest of the proof is the same for the remaining subgoals: here is the complete proof.
We now prove a simple induction theorem over the outputs.

The remaining theorems we prove about the outputs are ones describing their representation, distinctiveness and equality.

We now prove a theorem about the distinctiveness of each of the output terms. First, we create a list of theorems consisting of the invertibility
of each of the output representations. This theorem list will be useful in proving the theorem.

```plaintext

#let L = map (SYM o (REWRITE_RULE [ IS_output ]))
[SPEC "(Red, Yellow, T)" REP_decode_output;
 SPEC "(Red, Yellow, F)" REP_decode_output;
 SPEC "(Red, Green, T)" REP_decode_output;
 SPEC "(Red, Green, F)" REP_decode_output;
 SPEC "(Yellow, Red, T)" REP_decode_output;
 SPEC "(Yellow, Red, F)" REP_decode_output;
 SPEC "(Green, Red, T)" REP_decode_output;
 SPEC "(Green, Red, F)" REP_decode_output;
];
L =
[ |- REP_output(decode_output(Red,Yellow,T)) = Red,Yellow,T;
  |- REP_output(decode_output(Red,Yellow,F)) = Red,Yellow,F;
  |- REP_output(decode_output(Red,Green,T)) = Red,Green,T;
  |- REP_output(decode_output(Red,Green,F)) = Red,Green,F;
  |- REP_output(decode_output(Yellow,Red,T)) = Yellow,Red,T;
  |- REP_output(decode_output(Yellow,Red,F)) = Yellow,Red,F;
  |- REP_output(decode_output(Green,Red,T)) = Green,Red,T;
  |- REP_output(decode_output(Green,Red,F)) = Green,Red,F]
: thm list

Next, we set the actual goal.

#g " ~(GRF = GRT) /
   ~(GRF = YRF) /
   ~(GRF = YRT) /
   ~(GRF = YRF) /
   ~(GRF = RGF) /
   ~(GRF = RGT) /
   ~(GRF = RGF) /
   ~(GRF = YRT) /
   ~(GRF = YRF) /
   ~(YRF = YRT) /
   ~(YRF = RGF) /
   ~(YRF = RGF) /
   ~(YRF = RGT) /
   ~(YRF = RGF) /
   ~(YRF = RYT) /
   ~(YRF = YRF) /
   ~(YRF = RYT) /
   ~(RGF = RGT) /
   ~(RGF = RYF) /
   ~(RGF = RYT) /
   ~(RGF = RYF) /
   ~(RGF = RYF)
";;
```
We rewrite the goal by expanding using the definitions of each output term.
We simplify the goal to each of its conjuncts.

```
#e(REPEAT_CONJ_TAC);
OK.
25 subgoals

% << **** 27 subgoals omitted **** >> %

"(decode_output(Green,Red,F) = decode_output(Green,Red,T))"

() : void
```

We apply REP_output to each side of the equality.
We complete the proof by using the equality clauses for colour, the definition of what an output representation is, and the list of theorems, L.

The proof in tidy form appears below.
The inequality clauses for output are proved as follows

```plaintext
#let output_DISTINCT = prove_thm
('output_DISTINCT',
 "'(GRF = GRT) /
  '(GRF = YRF) /
  '(GRF = YRT) /
  '(GRF = RGF) /
  '(GRF = RGT) /
  '(GRF = RYF) /
  '(GRF = RYT) /
  '(GRT = YRF) /
  '(GRT = YRT) /
  '(GRT = RGF) /
  '(GRT = RGT) /
  '(YRF = YRT) /
  '(YRF = RGF) /
  '(YRF = RGT) /
  '(YRF = RYF) /
  '(YRF = RYT) /
  '(YRT = RGF) /
  '(YRT = RGT) /
  '(YRT = RYF) /
  '(YRT = RYT) /
  '(RGF = RGF) /
  '(RGF = RGT) /
  '(RGF = RYF) /
  '(RGF = RYT) /
  '(RGT = RGF) /
  '(RGT = RGT) /
  '(RGT = RYF) /
  '(RGT = RYT) /
  '(RYF = RGF) /
  '(RYF = RGT) /
  '(RYF = RYF) /
  '(RYF = RYT)",
PURE_REWRITE_TAC
[GRF; GRT; YRF; YRT; RGF; RGT; RYF; RYT]
THEN REPEAT CONJ_TAC
THEN DISCH_THEN
  \th. MP_TAC
(AP_TERM "REP_output:output->(colour#colour#bool)" th)
THEN REWRITE_TAC
([AP_TERM "IS_output:PAIR_EQ; colour_EQ_CLAUSES @ L"]
where L = map (SPEC o (REWRITE_RULE [IS_output]))
[SPEC "(Red, Yellow, T)" REP_decode_output;
 SPEC "(Red, Yellow, F)" REP_decode_output;
 SPEC "(Red, Green, T)" REP_decode_output;
 SPEC "(Red, Green, F)" REP_decode_output;
 SPEC "(Yellow, Red, T)" REP_decode_output;
 SPEC "(Yellow, Red, F)" REP_decode_output;
 SPEC "(Green, Red, T)" REP_decode_output;
 SPEC "(Green, Red, F)" REP_decode_output;]
);
output_DISTINCT = ...
```

The inequality clauses for output are proved as follows
\#let output_EQ_CLAUSES = save_thm
   (`output_EQ_CLAUSES',
    LIST_CONV (`thm_list
       @ (map CONV_RULE (SND_CONV_SYM_CONV) thm_list)))
where thm_list = CONJUNCTS output_DISTINCT;;
output_EQ_CLAUSES =
|- (`GRF = GRT) /
  (`YRF = YRF) /
  (`GRF = YRT) /
  (`YRF = RGF) /
  (`GRF = RGT) /
  (`YRF = RYT) /
  (`GRF = YRF) /
  (`GRF = YRT) /
  (`GRF = RGF) /
  (`GRF = RGT) /
  (`GRF = RYT) /
  (`GRF = RYT) /
  (`GRF = RFT) /
  (`GRF = RGF) /
  (`GRF = RGT) /
  (`GRF = RYT) /
  (`GRF = RYT) /
  (`GRF = RGT) /
  (`GRF = RYT) /
  (`GRF = RYT) /
  (`GRF = RFT) /
  (`GRF = RGT) /
  (`GRF = RYT) /
  (`GRF = RYT) /
  (`GRF = RFT) /
  (`GRF = RGT) /
  (`GRF = RYT) /
  (`GRF = RYT) /
  (`GRF = RFT) /
  (`GRF = RGT) /
  (`GRF = RYT) /
  (`GRF = RYT) /
  (`GRF = RFT) /
  (`GRF = RGT) /
  (`GRF = RYT) /
  (`GRF = RYT) /
  (`GRF = RFT) /
  (`GRF = RGT) /
  (`GRF = RYT) /
  (`GRF = RYT) /
  (`GRF = RFT) /
  (`GRF = RGT) /
  (`GRF = RYT) /
  (`GRF = RYT) /
  (`GRF = RFT) /
  (`GRF = RGT) /
  (`GRF = RYT) /
  (`GRF = RYT) /
  (`GRF = RFT) /
  (`GRF = RGT) /
  (`GRF = RYT) /
  (`GRF = RYT) /
  (`GRF = RFT) /
  (`GRF = RGT) /
  (`GRF = RYT) /
  (`GRF = RYT) /
  (`GRF = RFT) /
  (`GRF = RGT) /
  (`GRF = RYT) /
  (`GRF = RYT) /
  (`GRF = RFT) /
  (`GRF = RGT) /
  (`GRF = RYT) /
  (`GRF = RYT) /
  (`GRF = RFT) /
  (`GRF = RGT) /
  (`GRF = RYT) /
  (`GRF = RYT) /
  (`GRF = RFT) /
  (`GRF = RGT) /
  (`GRF = RYT) /
  (`GRF = RYT) /
  (`GRF = RFT) /
  (`GRF = RGT) /
  (`GRF = RYT) /
  (`GRF = RYT) /
  (`GRF = RFT) /
  (`GRF = RGT) /
  (`GRF = RYT) /
  (`GRF = RYT) /
  (`GRF = RFT) /
  (`GRF = RGT) /
  (`GRF = RYT) /
  (`GRF = RYT) /
  (`GRF = RFT) /
  (`GRF = RGT) /
  (`GRF = RYT) /
  (`GRF = RYT) /
  (`GRF = RFT) /
  (`GRF = RGT) /
  (`GRF = RYT) /
  (`GRF = RYT) /
  (`GRF = RFT) /
  (`GRF = RGT) /
  (`GRF = RYT) /
  (`GRF = RYT) /
  (`GRF = RFT) /
  (`GRF = RGT) /
  (`GRF = RYT) /
  (`GRF = RYT) /
  (`GRF = RFT) /
  (`GRF = RGT) /
  (`GRF = RYT) /
  (`GRF = RYT) /
  (`GRF = RFT) /
  (`GRF = RGT) /
  (`GRF = RYT) /
  (`GRF = RYT) /
  (`GRF = RFT) /
  (`GRF = RGT) /
  (`GRF = RYT) /
  (`GRF = RYT) /
  (`GRF = RFT) /
  (`GRF = RGT) /
  (`GRF = RYT) /
  (`GRF = RYT) /
  (`GRF = RFT) /
  (`GRF = RGT) /
  (`GRF = RYT) /
  (`GRF = RYT) /
  (`GRF = RFT) /
21.3.3 State transition and output function definitions

The state and output functions are simply defined in terms of the data types we have created. The state transition function is defined followed by the output function.

```plaintext
#let state_trans = new_definition
  ("state_trans ",
   "state_trans(s:state, (C:bool, T_L:bool, T_S:bool)) = ((s = HG) => ((C \ T_L) => HY | HG)
     | ((s = HY) => (T_S => PG | HY)
     | ((s = FG) => ((C \ T_L) => FY | FG)
     | (T_S => HG | FY)))
  ”);
state_trans = ...

#let output_table = new_definition
  ("output_table ",
   "output_table(s:state, (C:bool, T_L:bool, T_S:bool)) =
     ( (s = HG) => ( (C \ T_L) => GRT | GFL)
     | ( (s = HY) => ( T_S => YRT | YRF)
     | ( (s = FG) => ( (C \ T_L) => RGT | RGF)
     | ( T_S => RYT | RYF)))
  ”);
output_table = ...
```

21.3.4 Overview of the machine implementations

At this point we have several resources available to us.

2. A specification of behaviour for the controller.
3. A data type structure for the controller specification.

The data type structure introduced in the controller specification will be followed by both of our implementations. What we will do in each implementation is define a state encoding, output encoding, and input encoding. We will then relate these encodings to the specification. The encodings between each implementation will be shown to be invertible. Once this is shown, both implementations will be shown to be equivalent to the specification and to each other.

Machine M1 will use a gray code encoding of the states. Machine M2 will use a "one-hot" state encoding, (such encodings are convenient for field programmable gate array implementations). Both machines will use the same output encodings. Also, since the inputs are in binary form and
all tuples are valid, no data structure will be introduced for the inputs. We note that the state encoding used for M1 corresponds to the development in [73].

Since both machines use colour, we will encode traffic light colour with two bits. We will introduce a traffic colour data type and represent it as a binary pair. Both machines will use the same encoding. This encoding corresponds to [73].

The proofs in the following sections have forms which are very similar to the proofs done previously for the traffic light specification. In the interest of brevity will not provide much comment about them. To help, we have tried to use consistent names across the specification and both implementations. This should help in tracking down the original explanation for a proof of the first theorem of a similar class of theorems.

### 21.3.5 Creating the traffic colour data type

The coding we will use is $(F,F)$ for green, $(F,T)$ for yellow, and $(T,F)$ for red. We introduce these pairs as representations for the $M$ colour data type. Notice that we are not introducing them for the colour data type which already exists and has its own representation. Eventually, we will be able to show that everything is equivalent.

Similar to the definition of the output data type, we create a new data type with a concrete representation by first specifying the subset of the concrete representation to be used. We are using two bits to represent three colours. The one encoding which will not be used is $(T,T)$.

The definitions and theorems defined and proved for the representation of traffic light colours follows the same pattern of the output data type and is shown in the table below. Recall that within the role column, D means “definition” and P means “property”.

In precisely the same way we did for the output data type, we axiomatize the concrete representation for the machine colours.
To make the specific map between pairs and colours we define the `decode_Mcolour` function and use it to define machine colours MGreen, MYellow, and MRed.
21.3. DESCRIBING STATE MACHINES USING TYPES

decode_Mcolour =
| x. decode_Mcolour x = ABS_Mcolour(IS_Mcolour x => x | (T,F))

MGreen_DEF = | MGreen = decode_Mcolour(F,F)

MYellow_DEF = | MYellow = decode_Mcolour(F,T)

MRed_DEF = | MRed = decode_Mcolour(T,F)

In the same way as was done for the output data type, we prove the theorems which characterize the Mcolour data type.

IS_REP_Mcolour =| y. IS_Mcolour(REP_Mcolour y)

decode_REP_Mcolour = | y. decode_Mcolour(REP_Mcolour y) = y

Mcolour_Cases =| y. (y = MGreen) | (y = MYellow) | (y = MRed)

Mcolour_INduct =| y. P MGreen | P MYellow | P MRed => (!y. P y)

RE懿 decode_Mcolour =
| x. IS_Mcolour x = (x = REP_Mcolour(decode_Mcolour x))

Mcolour_DISTINCT =
| ~(MGreen = MYellow) | ~(MGreen = MRed) | ~(MYellow = MRed)

Mcolour_EQ_CLAUSEs =
| ~(MGreen = MYellow) /
| ~(MGreen = MRed) /
| ~(MYellow = MRed) /
| ~(MYellow = MGreen) /
| ~(MRed = MGreen) /
| ~(MRed = MYellow)

21.3.6 Machines M1 and M2

We begin the definitions for our implementations of the traffic light controller. Our ultimate objectives are

1. Derive entirely Boolean descriptions for the state transformation and output functions, and

2. Show that the behaviour of M1 and M2 based on the Boolean descriptions corresponds to the symbolic specification descriptions.

The first objective is easy given the framework we have developed for finite state machines. All the effort will be in proving the necessary invertibility theorems to show the second objective.
21.3.7 Data type definitions for M1

To define M1 we will introduce two data types, \texttt{Mstate} and \texttt{Moutput}. Once we have the necessary representation and decoding functions, we will be able to relate M1 to the specification. Data type \texttt{Mstate} is supported by a pair of Booleans. The coding we have in mind is M1HG = (F,F), M1HY = (F,T), M1FG = (T,T), and M1FY = (T,F). We introduce the predicate which defines the set of pairs to be used in the representation in the normal fashion. Since all possible Boolean pairs are included, the definition and proof of non-emptiness is trivial.

We axiomatize the concrete representation of the machine states in the same way as before.

\begin{verbatim}
IS_Mstate =
|! x. IS_Mstate x = (x = F,F) \ OR (x = F,T) \ OR (x = T,F) \ OR (x = T,T)

Mstate_Exists = |- ?x. IS_Mstate x

Mstate_TYP_DEF = |- ?rep. TYPE_DEFINITION IS_Mstate rep

Mstate_ISO_DEF =
|! (a. ABS_Mstate(REP_Mstate a) = a) \ OR (r. IS_Mstate r = (REP_Mstate(ABS_Mstate r) = r))

REP_Mstate_INVERTS = |- !a. ABS_Mstate(REP_Mstate a) = a

REP_Mstate_11 = |- !a a'. (REP_Mstate a = REP_Mstate a') = (a = a')

REP_Mstate_ONTO = |- !r. IS_Mstate r = (?a. r = REP_Mstate a)

ABS_Mstate_INVERTS =
|! r. IS_Mstate r = (REP_Mstate(IS_Mstate r) = r)

ABS_Mstate_11 =
|! r r'.
   IS_Mstate r ==> IS_Mstate r' ==> ((ABS_Mstate r = ABS_Mstate r') = (r = r'))

ABS_Mstate_ONTO = |- !a. ?r. (a = ABS_Mstate r) \ OR IS_Mstate r
\end{verbatim}

The decoding function for M1state is defined along with the actual association between the states and representations.
In the same way as was done for the previous concrete representations, we prove the theorems which characterize the `Mstate` data type.

```
decode_Mstate  =  |\, !x. decode_Mstate x = ABS_Mstate(IS_Mstate x => x | (F,F))
M1HG_DEF     =  |\, M1HG = decode_Mstate(F,F)
M1HY_DEF     =  |\, M1HY = decode_Mstate(F,T)
M1FY_DEF     =  |\, M1FY = decode_Mstate(T,F)
M1FG_DEF     =  |\, M1FG = decode_Mstate(T,T)
```

We introduce the data type `M1output` in the same way as we did for the output data type the specification.
First, we axiomatize the representation.

```
IS_Mioutput = 
|-> !x.
   IS_Mioutput x =
   (x = MGreen, MRed, F) \/
   (x = MGreen, MRed, T) \/
   (x = MYellow, MRed, F) \/
   (x = MYellow, MRed, T) \/
   (x = MRed, MGreen, F) \/
   (x = MRed, MGreen, T) \/
   (x = MRed, MYellow, F) \/
   (x = MRed, MYellow, T)

MIoutput_Exists = |-> ?x. IS_Mioutput x

MIoutput_TY_DEF = |-> ?rep. TYPE_DEFINITION IS_Mioutput rep

MIoutput_ISO_DEF = 
|-> (!a. ABS_Mioutput(REP_Mioutput a) = a) \/
   (!r. IS_Mioutput r = (REP_Mioutput(ABS_Mioutput r) = r))

REP_Mioutput_INVERTS = |-> !a. ABS_Mioutput(REP_Mioutput a) = a

REP_Mioutput_11 = 
|-> !a \ a'. (REP_Mioutput a = REP_Mioutput a') = (a = a')

REP_Mioutput_ISO = |-> !r. IS_Mioutput r = (?a. r = REP_Mioutput a)

ABS_Mioutput_INVERTS = 
|-> !r. IS_Mioutput r = (REP_Mioutput(ABS_Mioutput r) = r)

ABS_Mioutput_11 = 
|-> !r \ r'.
   IS_Mioutput r ==>
   IS_Mioutput r' ==>
   ((ABS_Mioutput r = ABS_Mioutput r') = (r = r'))

ABS_Mioutput_ISO = |-> !a \ ?r. (a = ABS_Mioutput r) \ IS_Mioutput r
```

We define the mapping between the output and its representation in exactly the same way as before.
In the same way as was done for the previous data types with concrete representations, we prove the theorems which characterize the \texttt{M1output} data type.
CHAPTER 21. VERIFICATION OF FSMS

```
IS_REP_M1output = \= y. IS_M1output(REP_M1output y)

decode_REP_M1output = \= y. decode_M1output(REP_M1output y) = y

M1output_Cases =
\= y.
  (y = M1GRF) /
  (y = M1GRT) /
  (y = M1YRF) /
  (y = M1YRT) /
  (y = M1RGF) /
  (y = M1RGT) /
  (y = M1RYF) /
  (y = M1RYT)

M1output_Induct =
\= P.
  P M1GRF /
  P M1GRT /
  P M1YRF /
  P M1YRT /
  P M1RGF /
  P M1RGT /
  P M1RYF /
  P M1RYT =>>
  (y: P y)

REP_decode_M1output =
\= x. IS_M1output x = (x = REP_M1output(decode_M1output x))

M1output_DISTINCT =
\= (M1GRF = M1GRT) /
  "(M1GRF = M1YRF) /
  "(M1GRF = M1YRT) /
  "(M1GRF = M1RGF) /
  "(M1GRF = M1RGT) /
  "(M1GRF = M1RYF) /
  "(M1GRF = M1RYT) /
  "(M1GRT = M1YRF) /
  "(M1GRT = M1YRT) /
  "(M1GRT = M1RGF) /
  "(M1GRT = M1RGT) /
  "(M1GRT = M1RYF) /
  "(M1GRT = M1RYT) /
  "(M1YRF = M1YRT) /
  "(M1YRF = M1RGF) /
  "(M1YRF = M1RGT) /
  "(M1YRF = M1RYF) /
  "(M1YRF = M1RYT) /
  "(M1YRT = M1RGF) /
  "(M1YRT = M1RGT) /
  "(M1YRT = M1RYF) /
  "(M1YRT = M1RYT) /
  "(M1RGF = M1RGT) /
  "(M1RGF = M1RYF) /
  "(M1RGF = M1RYT) /
  "(M1RGT = M1RYF) /
  "(M1RGT = M1RYT) /
  "(M1RYF = M1RYT) /
```

\text{21.3. \textsc{Describing State Machines Using Types}}

\begin{verbatim}
M1 output_EQ_CLAUSES =
| - (M1GRF = M1GRT) /
| - (M1GRF = M1YRF) /
| - (M1GRF = M1YRT) /
| - (M1GRF = M1RGT) /
| - (M1GRF = M1RGF) /
| - (M1GRF = M1RYF) /
| - (M1GRF = M1RYT) /
| - (M1GRT = M1YRF) /
| - (M1GRT = M1YRT) /
| - (M1GRT = M1RGT) /
| - (M1GRT = M1RGF) /
| - (M1GRT = M1RYF) /
| - (M1GRT = M1RYT) /
| - (M1YRF = M1YRT) /
| - (M1YRF = M1RGF) /
| - (M1YRF = M1RGT) /
| - (M1YRF = M1RYF) /
| - (M1YRF = M1RYT) /
| - (M1YRT = M1RGF) /
| - (M1YRT = M1RGT) /
| - (M1YRT = M1RYF) /
| - (M1YRT = M1RYT) /
| - (M1RGT = M1RGF) /
| - (M1RGT = M1RGT) /
| - (M1RGT = M1RGF) /
| - (M1RGT = M1RYF) /
| - (M1RGT = M1RYT) /
| - (M1RYF = M1YRT) /
| - (M1RYF = M1RGT) /
| - (M1RYF = M1RYF) /
| - (M1RYF = M1RYT) /
| - (M1RYF = M1RYT)
\end{verbatim}
21.3.8  Data type definitions for M2

Machine M2 has a different state representation than does M1. M2 uses a one-hot state encoding. In all other respects it is the same. The introduction of the state representation for M2 is virtually the same as it was for M1.

We first axiomatize the representation for $\text{M2state}$.

\begin{verbatim}
IS_M2state =  \
  |!x. IS_M2state x =  \
   (x = T,F,F,F) \lor (x = F,T,F,F) \lor (x = F,F,T,F) \lor (x = F,F,F,T) \\
M2state_Exists = |- ?x. IS_M2state x \\
M2state_TYP_DEF = |- ?rep. TYPE_DEFINITION IS_M2state rep \\
M2state_ISO_DEF =  \
  |- (!a. ABS_M2state(REP_M2state a) = a) /\  
  (r. IS_M2state r = (REP_M2state(ABS_M2state r) = r)) \\
REP_M2state_INVERTS = |- !a. ABS_M2state(REP_M2state a) = a \\
REP_M2state_11 = |- !a a'. (REP_M2state a = REP_M2state a') = (a = a') \\
REP_M2state_OTTO = |- !r. IS_M2state r = (?a. r = REP_M2state a) \\
ABS_M2state_INVERTS =  \
  |- !r. IS_M2state r = (REP_M2state(ABS_M2state r) = r) \\
ABS_M2state_11 =  \
  |- !r r'.  
   IS_M2state r =>  
   IS_M2state r' =>  
   (ABS_M2state r = ABS_M2state r') = (r = r')) \\
ABS_M2state_OTTO = |- !a. ?r. (a = ABS_M2state r) /\ IS_M2state r \\
\end{verbatim}

We define the mapping between $\text{M2state}$ and its representation.

\begin{verbatim}
decode_M2state =  \
  |- !x. decode_M2state x = ABS_M2state(IS_M2state x => x | (F,F,F,T)) \\
M2HG_DEF = |- M2HG = decode_M2state(T,F,F,F) \\
M2HY_DEF = |- M2HY = decode_M2state(F,T,F,F) \\
M2FG_DEF = |- M2FG = decode_M2state(F,F,T,F) \\
M2FY_DEF = |- M2FY = decode_M2state(F,F,F,T)
\end{verbatim}
Finally, we prove the theorems which characterize \( \text{M2state} \) in the same way as in the other data types with concrete representations.

\[
\begin{align*}
\text{IS\_REP\_M2state} &= \lambda y. \text{IS\_M2state}(\text{REP\_M2state} y) \\
\text{decode\_REP\_M2state} &= \lambda y. \text{decode\_M2state}(\text{REP\_M2state} y) = y \\
\text{M2state\_Cases} &= \lambda y. (y = \text{M2HG}) \lor (y = \text{M2HY}) \lor (y = \text{M2FG}) \lor (y = \text{M2FY}) \\
\text{M2state\_Induct} &= \lambda P. P \text{M2HG} \lor P \text{M2HY} \lor P \text{M2FG} \lor P \text{M2FY} => (\lambda y. P y) \\
\text{REP\_decode\_M2state} &= \lambda x. \text{IS\_M2state} x = (x = \text{REP\_M2state}(\text{decode\_M2state} x)) \\
\text{M2state\_DISTINCT} &= \lambda (\text{M2HG} = \text{M2HY}) \lor \\
&\quad (\text{M2HG} = \text{M2FG}) \lor \\
&\quad (\text{M2HG} = \text{M2FY}) \lor \\
&\quad (\text{M2HY} = \text{M2FG}) \lor \\
&\quad (\text{M2HY} = \text{M2FY}) \lor \\
&\quad (\text{M2FY} = \text{M2FG}) \lor \\
&\quad (\text{M2FY} = \text{M2FY}) \\
\text{M2state\_EQ\_CLAUSES} &= \lambda (\text{M2HG} = \text{M2HY}) \lor \\
&\quad (\text{M2HG} = \text{M2FG}) \lor \\
&\quad (\text{M2HG} = \text{M2FY}) \lor \\
&\quad (\text{M2HY} = \text{M2FG}) \lor \\
&\quad (\text{M2HY} = \text{M2FY}) \lor \\
&\quad (\text{M2FY} = \text{M2FG}) \lor \\
&\quad (\text{M2FY} = \text{M2FY}) \lor \\
&\quad (\text{M2FY} = \text{M2HG}) \lor \\
&\quad (\text{M2FY} = \text{M2HG}) \lor \\
&\quad (\text{M2FY} = \text{M2HG}) \lor \\
&\quad (\text{M2FY} = \text{M2HG})
\end{align*}
\]

21.3.9 Relating M1 and M2 to the specification

Verification of M1

We now proceed with the verification of M1. To do this we must define the mapping functions which get us from the specification to M1 and back again.

To help us with the proofs and to make the terms succinct, we bind several ml names to HOL types.
Encoding and Decoding Functions

The first mapping function we define moves us from states of the specification to states of M1.

```ocaml
#let S2M1S = new_definition
  ('S2M1S',
   "S2M1S (s:state)
    = (COND (s = HG) M1HG
        (COND (s = HY) M1HY
           (COND (s = FG) M1FG
              M1FY))));;
S2M1S = ...```

Using S2M1S and the REP_M1state function will get us from states of the specification to the Boolean representations of the states of M1.

```ocaml
#let encode_S1 = new_definition
  ('encode_S1',
   "encode_S1 = REP_M1state o S2M1S");;
encode_S1 = ...```

To get us back from the Boolean representations of the states of M1 to the states of the specifications we define the inverses of the encoding functions.
We also define the encoding and decoding functions for the inputs to the specification and M1. Since the inputs are just Booleans, the mapping functions are just the identity function, I. Recall that we need these functions as parameters for the correctness theorems.

In a similar fashion to the way we defined the state mappings, we define the output mappings to and from the specification outputs and the outputs of M1.
CHAPTER 21. VERIFICATION OF FSMS

Recall that the representation of the M outputs were in terms of M colour. Since we want Boolean representations only, we must specify how we will get from 3-tuples containing elements of M colour to 3-tuples having only Boolean structures. This is done below by REP_M10bools. Its inverse bools2REP_M10 is also defined.
We now have all the functions necessary to get us from outputs at the specification level to Boolean representations of outputs in M1. We define the encoding and decoding functions as shown.

```ocaml
#let encode_01 = new_definition
  ("encode_01",
   "encode_01 = REP_M10/2bools o REP_M1output o 02M101");;
encode_01 = ...

#let decode_01 = new_definition
  ("decode_01",
   "decode_01 = M1020 o decode_M1output o bools3REP_M101");;
decode_01 = ...
```

### State-transition and Output Functions

Having defined the encoding and decoding functions we can immediately generate their Boolean equivalents using M_state_trans and M_output_table.

```ocaml
#let M1_state_trans = new_definition
  ("M1_state_trans",
   "M1_state_trans ((s:"bM1state),(x:"input)) =
    M_state_trans state_trans encode_S1 decode_S1 decode_I1 (s,x)");;
M1_state_trans = ...

#let M1_output_table = new_definition
  ("M1_output_table",
   "M1_output_table (s:"bM1state,x:"input) =
    M_output_table output_table encode_01 decode_S1 decode_I1 (s,x)");;
M1_output_table = ...
```

### Equivalence Proofs

We now must show that the encoding and decoding functions invert.

We first start with the state encoding and decoding functions by showing that \( \text{decode}_S1 \circ \text{encode}_S1 = I \).

```ocaml
#let lemma1 = PROVE
  ("!:s:state.(M1S2S o S2M1S) s = s",
   INDUCT_theorem state_induct ASSERT_TAC
   THEN PURE_REWRITE_TAC [ o_DEF; M1S2S; S2M1S]
   THEN CONV_TAC (DEPTH_CONV Beta_CONV)
   THEN REWRITE_TAC [ state_EQ_CLAUSES; M1state_EQ_CLAUSES ]);
lemma1 = ...
```

The remaining lemmas show that invertibility of M1S2S and S2M1S.
Next, we prove that \texttt{decode} \_\texttt{S1} and \texttt{encode} \_\texttt{S1} invert each other.

\begin{verbatim}
#let lemma5 = EXT lemma4;; lemma5 = | decode_Mstate o REP_Mstate = I

#let o_LEMMA1 = SYM (SPEC_ALL o_THM);; o_LEMMA1 = | - f(g x) = (f o g)x

#let o_ASSOC_LEMMA1 = PROVE
  "(!f1:*->*)(f2:***->*)(g1:***->*)(g2:***->**).
  (f1 o f2) o (g1 o g2) = f1 o ((f2 o g1) o g2",
  REWRITE_TAC [ o_ASSOC ]);; o_ASSOC_LEMMA1 = 
  | - !f1 f2 g1 g2. (f1 o f2) o (g1 o g2) = f1 o ((f2 o g1) o g2)
\end{verbatim}

We expand using the definitions.

\begin{verbatim}
# (PURE_ONCE_REWRITE_TAC [ decode_S1; encode_S1]);; OK.

"!s. (MSTATE o SEMIS) ((REP_Mstate o SEMIS)s) = s"

() : void
\end{verbatim}

Next, we reassociate the terms of the subgoal.
Next, we rewrite using the invertibility lemmas and the properties of $I$.

The proof in tidy form is shown below.

Since the input is accepted "as is", it is easy to show that the encoding and decoding of the input is an identity.
Finally, we move on to showing the output mappings invert one another.

The next two theorems show the composition of the mapping functions is I.

Next, we show that the mapping to and from the Boolean representation is correct.

The following lemmas are extensions of lemma9.
Finally, with the above lemmas, we can show that `decode_01` inverts `encode_01`.

```plaintext
#g "!out. decode_01(encode_01 out) = out";
"!out. decode_01(encode_01 out) = out"

() : void
```

First, we expand using the definitions.

```plaintext
#e(PURE_REWRITE_TAC [ decode_01; encode_01]);;
OK.
"!out. (!M1020 o (decode_M1output o bools_REP_M10))
((REP_M102bools o (REP_M1output o OEM10))out) = out"

() : void
```

We complete the proof by a series of associations.
#e(PURE_REWRITE_TAC
    [INST_TYPE
     ['"output","**"';""";'"bM1out","**"';""";'"output","***";]
     o_LEMMMA1]);

OK.

"!out.
((M120 o (decode_M1output o bools2REP_M10)) o
 (REP_M102bools o (REP_M1output o 02M10)))
out =
out"

(): void

#e(PURE_REWRITE_TAC
    [INST_TYPE
     ['"M1output","***"';""";'"output","**"';""";'"bM1out","**";
     ("";'"bM1out","**")]
     o_ASSOC];

OK.

"!out.
((M120 o decode_M1output) o
 (bools2REP_M10 o REP_M102bools) o (REP_M1output o 02M10))
out =
out"

(): void

#e(PURE_REWRITE_TAC [o_ASSOC_LEMMMA1]);

OK.

"!out. ((M120 o decode_M1output) o (REP_M1output o 02M10))out = out"

(): void

#e(PURE_REWRITE_TAC [lemma12;I.o_ID]);

OK.

"!out. ((M120 o decode_M1output) o (REP_M1output o 02M10))out = out"

(): void

#e(PURE_REWRITE_TAC [o_ASSOC_LEMMMA1]);

OK.

"!out. (M120 o ((decode_M1output o REP_M1output) o 02M10))out = out"

(): void

#e(REWRITE_TAC [lemma14;lemma7;I.o_ID;I_THM]);

OK.

goal proved

% << **** trace omitted **** >> %

Previous subproof:
goal proved

(): void
The proof in tidy form appears below.

```
#let decode_encode_O1 = prove_thm
  ('decode_encode_O1',
   "!out. decode_O1(encode_O1 out) = out",
   PURE_REWRITE_TAC [ decode_O1; encode_O1]
   THEN PURE_REWRITE_TAC
   [ INST_TYPE
     [("";"out";;"*";;:"out";;"*")]
     o_LEMM1]
   THEN PURE_REWRITE_TAC
   [ INST_TYPE
     [ ["":"output";;"*";;:"*";("":"output";;"*")]
     o_ASSOC]
   THEN PURE_REWRITE_TAC [ o_ASSOC_LEMM1]
   THEN PURE_REWRITE_TAC [ lemma2; I_o_ID]
   THEN PURE_REWRITE_TAC [ o_ASSOC_LEMM1]
   THEN REWRITE_TAC [ lemma14; lemma7; I_o_ID; I_THM];
   decode_encode_O1 = !out. decode_O1(encode_O1 out) = out
```

Now that all the mapping function have been shown to be correct, we can prove the correctness of the state transition and output functions by just substituting into the appropriate theorems and doing modus ponens and rewriting when needed.

```
#let M1_state_trans_lemma
  = (MP (ISPECL
    ["encode_S1:state->"bMstate";decode_S1:"bMstate->state";
     "encode_I1:input->"input";"decode_I1:"input->"input";
     "state_trans:state # input->state"] M_state_trans_correct)
   (COND decode_encode_S1 decode_encode_I1));
M1_state_trans_lemma = !s x. encode_S1(state_trans(s,x)) = M_state_trans
                      state_trans
                      encode_S1
                      decode_S1
                      decode_I1
                      (encode_S1 s,encode_I1 x)
```

We now prove the correctness theorem for the state transition function which shows the isomorphic relationship between state_trans and M1_state_trans.
Now, we proceed to prove the correctness of the implementation of the output function. First, we prove a lemma about the output table.
We use the correctness of the state transition and output implementations and the parametric correctness theorems proved earlier for all state machines, to prove more correctness theorems. First, we show that state transition implementation is correct for any sequence of inputs and starting states.

```
#let M1_state_match_thm = save_thm
   ('M1_state_match_thm',
    GEN_ALL,
    (MP
     (REWRITE_RULE
      []
       (ISPECL
        ['inlist:("input")";"s":state";"(encode_S1:state->bM1state)(s":state)"
         "MAP (encode_I1:"input->"input)(inlist:"input")"
         "encode_S1:state->bM1state";"encode_I1:input->"input"
         "state_trans:state#"input->state"
         "M1_state_trans:bM1state#"input->"bM1state"
         "decode_S1:"bM1state->state";"decode_I1:"input->"input"
         ) state_match_thm))
    (CONJ decode_encode_S1 M1_state_trans_correct));
M1_state_match_thm
 |- !inlist s.
   eval_S state_trans inlist s =
   decode_S1(eval_S M1_state_trans(MAP encode_I1 inlist)(encode_S1 s))
```

Next, we show that the implementation of the output function is correct for any sequence of inputs and starting states.
We can also show the mappings between the state transition behaviour and implementation are correct.

Lastly, we show the correctness of the output function mapping.
21.3. DESCRIBING STATE MACHINES USING TYPES

#let M1_output_map_correct  
  = (REWRITE_RULE [SYM(ISPEC  
                          ["decode_S1: bM1_output->output";  "M1_output_table: bM1state->input->bM1out";  
                          "x:"bM1state" input"  o_THM]) M1_output_table_correct]);
M1_output_map_correct =  
| s x.  
output_table(s,x) =  
(decode_S1 o M1_output_table)(encode_S1 s,encode_L1 x)

Verification of M2

To verify M2 all we need do is define the proper mapping functions to and from states of the specification and states of M2. The input and output mapping functions remain the same.

To simplify the appearance of the terms, we bind ML variables to HOL types.

#let bM2state = ":bool#bool#bool#bool"  
and bMcolour = ":bool#bool";;
bM2state = ":bool # (bool # (bool # bool)) : type  
bMcolour = ":bool # bool" : type

#let bM1out = ":Mcolour#Mcolour#bool"  
and bM1out = ":bMcolour#bMcolour#bMcolour#bool"  
and input = ":bM1out#bool";;  
bM1out = ":Mcolour # (Mcolour # bool) : type  
bM1out = ":(bMcolour # bMcolour# bMcolour) # (bMcolour # bMcolour # bool) : type  
input = ":bool # (bool # bool)" : type

The mapping functions to and from states of the specification to states of M2 are shown below.
Having defined the state mapping functions we can derive the Boolean state-transition and output functions.

As before we show that decode\(_{M\text{state}}\) o REP\(_{M\text{state}}\) = I.

Next, we prove that decode\(_S2\) inverts encode\(_S2\).
21.3. **Describing State Machines Using Types**

We now prove the correctness theorems for the state transition and output function implementations.

\[
\text{decode} \_\text{encode} \_S \_2 = !s. \ \text{decode} \_S \_2 (\text{encode} \_S \_2 s) = s
\]

Relating \( M/1 \) and \( M/2 \)

Having related both \( M/1 \) and \( M/2 \) to the same specification, it is trivial to relate them to each other. We note that even though \( M/1 \) and \( M/2 \) have different underlying state representations, we can easily show their behaviour is equivalent. First, we show the state transition and output functions are the same.
Next, we show that for any sequence of inputs and starting states, \( M_1 \) and \( M_2 \) behave equivalently.

\[
\begin{align*}
\text{let } &M_1.M_2.\text{state_map_equal} = \text{REWWRITE_RULE [M1.state_map_correct] M2.state_map_correct}; \\
&M_1.M_2.\text{state_map_equal} = \mu!s x. \\
&(\text{decode}_S\circ\text{M1.state_trans})(\text{encode}_S s,\text{encode}_I x) = \\
&(\text{decode}_S\circ\text{M2.state_trans})(\text{encode}_S s,\text{encode}_I x) \\
\text{let } &M_1.M_2.\text{output_table_equal} = \text{REWWRITE_RULE [M1.output_table_correct] M2.output_table_correct}; \\
&M_1.M_2.\text{output_table_equal} = \mu!s x. \\
&(\text{decode}_O\circ\text{M1.output_table})(\text{encode}_S s,\text{encode}_I x) = \\
&(\text{decode}_O\circ\text{M2.output_table})(\text{encode}_S s,\text{encode}_I x) \\
\text{let } &M_1.M_2.\text{state_match_thm} = \text{REWWRITE_RULE [M1.state_match_thm] M2.state_match_thm}; \\
&M_1.M_2.\text{state_match_thm} = \mu!s x. \\
&(\text{encode}_S\circ\text{eval}_S\circ\text{M1.state_trans})(\text{MAP encode}_I\circ\text{inlist})(\text{encode}_S s)) = \\
&(\text{encode}_S\circ\text{eval}_S\circ\text{M2.state_trans})(\text{MAP encode}_I\circ\text{inlist})(\text{encode}_S s)) \\
\text{let } &M_1.M_2.\text{outputs_match_thm} = \text{REWWRITE_RULE [M1.outputs_match_thm] M2.outputs_match_thm}; \\
&M_1.M_2.\text{outputs_match_thm} = \mu!s x. \\
&(\text{MAP decode}_O\circ\text{eval}_O\circ\text{M1.state_trans}\circ\text{M1.output_table})(\text{encode}_S s) \\
&(\text{MAP encode}_I\circ\text{inlist})) = \\
&(\text{MAP decode}_O\circ\text{eval}_O\circ\text{M2.state_trans}\circ\text{M2.output_table})(\text{encode}_S s) \\
&(\text{MAP encode}_I\circ\text{inlist}))
\end{align*}
\]
EXERCISES 21

**Exercise 21.1** Prove the theorems for the \texttt{Mcolour} data type.

**Exercise 21.2** Prove the theorems for the \texttt{Mstate} data type.

**Exercise 21.3** Prove the theorems for the \texttt{Moutput} data type.

**Exercise 21.4** Prove the theorems for the \texttt{M2state} data type.

**Exercise 21.5** Prove the theorems showing the correctness of M2.
Coda

To be the ‘compleat’ hardware verifier requires a solid background in formal computer science, mathematical logic, proof techniques, mechanized reasoning, and hardware design. We do not attempt to cover the entire field in this text—all we offer are the basics of specification, abstraction, and verification using the HOL system.

In other words, the text will help you through your apprenticeship in specifying and verifying hardware systems in HOL, but it will not turn you into a HOL wizard overnight. That requires practice and a broader perspective. However this text will give you the level of HOL maturity required to read through and learn from the proofs of others.

Well documented examples include [26, 27, 29, 34, 35, 65] and the examples by Gunter and Joyce in [107]. It is also worthwhile to study the source code for the HOL system which contains the proofs of many theorems by experienced HOL users. [10, 11, 24, 25, 62, 70, 81], are proceedings of hardware verification workshops and cover a broad spectrum of recent research in the area. You can get a more rounded understanding of higher order logic and natural deduction by studying the formal logic in [1] and Paulson’s text on LCF [90] respectively.

Other interesting theorem provers are AFFIRM [85], Boyer-Moore [13], M-EVES [87], IPL [95], Isabelle [89, 91], VERIFY [4], and Veritas [50, 51].

In the main, we have restricted ourselves to gate level proofs. For accounts of some delightful work going on below that level, and on relating the two levels, see [52, 76, 86, 114].
Appendix A

HOL built-ins

A.1 Axioms

BOOL_CASES_AX  \( \vdash t \cdot (t = T) \lor (t = F) \)
IMP_ANTISYM_AX  \( \vdash t_1 \cdot t_2 \cdot (t_1 \Rightarrow t_2) \Rightarrow (t_2 \Rightarrow t_1) \Rightarrow (t_1 = t_2) \)
ETA_AX  \( \vdash ! \cdot t \cdot (\forall x \cdot t x) = t \)
SELECT_AX  \( \vdash P x \cdot P x \Rightarrow p (\$@ P) \)
INFINITY_AX  \( \vdash ! f \cdot (\text{ONE_ONE} f) \lor (~\text{ONTU} f) \)

A.2 Primitive inference rules

ABS: term \( \rightarrow \) thm \( \rightarrow \) thm

"x"

\( A \vdash t_1 = t_2 \)

------------------------------------------------------
\( A \vdash (\forall x \cdot t_1) = (\forall x \cdot t_2) \)

ASSUME: term \( \rightarrow \) thm

"t"

\( A \vdash t \)

----------------------
\( \vdash t \)

BETA_CONV: term \( \rightarrow \) thm

"(\( x \cdot \text{body} \)) arg"

---------------------------------------------
\( \vdash (\( x \cdot \text{body} \)) arg = [ \text{body} ] (\text{arg} / x ) \)

DISCH: term \( \rightarrow \) thm \( \rightarrow \) thm

"t"

\( A \vdash th \)

--------
\( A \vdash t \Rightarrow th \)

INST_TYPE: (type \# type) list \( \rightarrow \) thm \( \rightarrow \) thm

\( [ (ty_1, TY_1); (ty_2, TY_2); \ldots; (ty_n, TY_n); ] \)

\( A \vdash th \)

--------
\( A \vdash [ \text{th} ] (ty_1 ty_2 \ldots ty_n / TY_1 TY_2 \ldots TY_n) \)
A.3 Manipulating terms

Data type: \( \text{term} = \text{string} \# \text{type} | \text{string} \# \text{type} \ | \text{term} \# \text{term} | \text{term} \# \text{term} \)

Making terms

<table>
<thead>
<tr>
<th>Cat. of term</th>
<th>Function and type</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>\text{mk}_\text{const}:((\text{string} # \text{type}) \to \text{term})</td>
</tr>
<tr>
<td>variable</td>
<td>\text{mk}_\text{var} :((\text{string} # \text{type}) \to \text{term})</td>
</tr>
<tr>
<td>application</td>
<td>\text{mk}_\text{comb} :((\text{term} # \text{term}) \to \text{term})</td>
</tr>
<tr>
<td>abstraction</td>
<td>\text{mk}_\text{abs} :((\text{term} # \text{term}) \to \text{term})</td>
</tr>
<tr>
<td>negation</td>
<td>\text{mk}_\text{neg} :((\text{term} # \text{term}) \to \text{term})</td>
</tr>
<tr>
<td>conjunction</td>
<td>\text{mk}_\text{conj} :((\text{term} # \text{term}) \to \text{term})</td>
</tr>
<tr>
<td>disjunction</td>
<td>\text{mk}_\text{disj} :((\text{term} # \text{term}) \to \text{term})</td>
</tr>
<tr>
<td>implication</td>
<td>\text{mk}_\text{imp} :((\text{term} # \text{term}) \to \text{term})</td>
</tr>
<tr>
<td>equivalence</td>
<td>\text{mk}_\text{iff} :((\text{term} # \text{term}) \to \text{term})</td>
</tr>
<tr>
<td>equal</td>
<td>\text{mk}_\text{eq} :((\text{term} # \text{term}) \to \text{term})</td>
</tr>
<tr>
<td>(\forall)</td>
<td>\text{mk}_\text{forall} :((\text{term} # \text{term}) \to \text{term})</td>
</tr>
<tr>
<td>(\exists)</td>
<td>\text{mk}_\text{exists} :((\text{term} # \text{term}) \to \text{term})</td>
</tr>
<tr>
<td>(\epsilon)</td>
<td>\text{mk}_\text{select} :((\text{term} # \text{term}) \to \text{term})</td>
</tr>
<tr>
<td>conditional</td>
<td>\text{mk}_\text{cond} :((\text{term} # \text{term}) \to \text{term})</td>
</tr>
<tr>
<td>let</td>
<td>\text{mk}_\text{let} :((\text{term} # \text{term} # \text{term}) \to \text{term})</td>
</tr>
</tbody>
</table>
Taking terms apart

<table>
<thead>
<tr>
<th>Cat. of term</th>
<th>Function and type</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>dest_const : (term -&gt; (string # type))</td>
</tr>
<tr>
<td>variable</td>
<td>dest_var : (term -&gt; (string # type))</td>
</tr>
<tr>
<td>application</td>
<td>dest_comb : (term -&gt; (term # term))</td>
</tr>
<tr>
<td>rator</td>
<td>(term -&gt; term)</td>
</tr>
<tr>
<td>rand</td>
<td>(term -&gt; term)</td>
</tr>
<tr>
<td>abstraction</td>
<td>dest_abs : (term -&gt; (term # term))</td>
</tr>
<tr>
<td>negation</td>
<td>dest_neg : (term -&gt; (term # term))</td>
</tr>
<tr>
<td>conjunction</td>
<td>dest_conj : (term -&gt; (term # term))</td>
</tr>
<tr>
<td>disjunction</td>
<td>dest_disj : (term -&gt; (term # term))</td>
</tr>
<tr>
<td>implication</td>
<td>dest_imp : (term -&gt; (term # term))</td>
</tr>
<tr>
<td>equivalence</td>
<td>dest_iff : (term -&gt; (term # term))</td>
</tr>
<tr>
<td>equal</td>
<td>dest_eq : (term -&gt; (term # term))</td>
</tr>
<tr>
<td>∀</td>
<td>dest_exists : (term -&gt; (term # term))</td>
</tr>
<tr>
<td>∃</td>
<td>dest_forall : (term -&gt; (term # term))</td>
</tr>
<tr>
<td>ε</td>
<td>dest_select : (term -&gt; (term # term))</td>
</tr>
<tr>
<td>conditional</td>
<td>dest_cond : (term -&gt; (term # term # term))</td>
</tr>
<tr>
<td>let</td>
<td>dest_let : (term -&gt; (term # term # term))</td>
</tr>
</tbody>
</table>

Querying terms

<table>
<thead>
<tr>
<th>Cat. of term</th>
<th>Function and type</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>is_const : (term -&gt; bool)</td>
</tr>
<tr>
<td>variable</td>
<td>is_var : (term -&gt; bool)</td>
</tr>
<tr>
<td>application</td>
<td>is_comb : (term -&gt; bool)</td>
</tr>
<tr>
<td>abstraction</td>
<td>is_abs : (term -&gt; bool)</td>
</tr>
<tr>
<td>negation</td>
<td>is_neg : (term -&gt; bool)</td>
</tr>
<tr>
<td>conjunction</td>
<td>is_conj : (term -&gt; bool)</td>
</tr>
<tr>
<td>disjunction</td>
<td>is_disj : (term -&gt; bool)</td>
</tr>
<tr>
<td>implication</td>
<td>is_imp : (term -&gt; bool)</td>
</tr>
<tr>
<td>equivalence</td>
<td>is_iff : (term -&gt; bool)</td>
</tr>
<tr>
<td>equal</td>
<td>is_eq : (term -&gt; bool)</td>
</tr>
<tr>
<td>∀</td>
<td>is_forall : (term -&gt; bool)</td>
</tr>
<tr>
<td>∃</td>
<td>is_exists : (term -&gt; bool)</td>
</tr>
<tr>
<td>ε</td>
<td>is_select : (term -&gt; bool)</td>
</tr>
<tr>
<td>conditional</td>
<td>is_cond : (term -&gt; bool)</td>
</tr>
<tr>
<td>let</td>
<td>is_let : (term -&gt; bool)</td>
</tr>
</tbody>
</table>
A.4 Manipulating theorems

Data type: `thm = term list # term`

Making theorems

Theorems are to be proved not faked. There should be no `mk_thm` — but there is.

Taking theorems apart

<table>
<thead>
<tr>
<th>Cat. of term</th>
<th>Function and type</th>
</tr>
</thead>
<tbody>
<tr>
<td>theorem</td>
<td><code>dest_thm : (thm -&gt; (term list # term))</code></td>
</tr>
<tr>
<td>hyp</td>
<td><code>hyp : (thm -&gt; term list)</code></td>
</tr>
<tr>
<td>concl</td>
<td><code>concl : (thm -&gt; term)</code></td>
</tr>
</tbody>
</table>

Querying theorems

There is no built-in query function `is_thm`, but it is easy to write:

```plaintext
#let is_thm th = (let (hyp, concl) = dest_thm th in true) ? false;;
```

A.5 Basic rewrite rules

```plaintext
#ABS_SIMP;;
|~ !t1 t2. (\x. t1) t2 = t1

#AND_CLAUSES;;
|~ !t.
   (T \ t = t) \/
   (t \ T = t) \/
   (F \ t = F) \/
   (t \ F = F) \/
   (t \ t = t)

#COND_CLAUSES;;
|~ !t1 t2. ((T => t1 | t2) = t1) \/
           ((F => t1 | t2) = t2)

#EQ_CLAUSES;;
|~ !t.
   ((T = t) = t) \/
   ((t = T) = t) \/
   ((F = t) = ~t) \/
   ((t = F) = ~t)
```
#EXISTS_SIMP;;
|- !t. (\x. t) = t

#(FST, SND);;
(\- !x y. FST(x,y) = x, \- !x y. SND(x,y) = y) : (thm # thm)

#FORALL_SIMP;;
|- !t. (\x. t) = t

#IFF_EQ;;
|- !t1 t2. t1 <-> t2 = (t1 = t2)

#IMP_CLAUSES;;
|- !t.
  (T \rightarrow t = t) /
  (t \rightarrow T = T) /
  (F \rightarrow t = t) /
  (t \rightarrow t = T) /
  (t \rightarrow F = \neg t)

#NOT_CLAUSES;;
|- (!t. \neg t = t) /
  (T \rightarrow \neg T = F) /
  (F \rightarrow \neg F = T)

#OR_CLAUSES;;
|- !t.
  (T \or t = T) /
  (t \or T = T) /
  (F \or t = t) /
  (t \or F = t) /
  (t \or t = t)

#PAIR;;
|- !x. FST x, SND x = x

#REFL_CLAUSE;;
|- !t.
  (x \rightarrow x = T)
Appendix B

Extras

B.1 ML code

let CANCEL_CONJ_TAC: tactic =
  let rec extr cncl tm =
    if (is_conj tm) then
      let l, r = dest_conj tm in
      (let ltm = extr cncl_l in mk_conj(ltm, extr cncl_r) ? ltm) ?
      (extr cncl_r) else
      if (exists (aconv tm) cncl) then fail else tm in
  let rec prv cs tm =
    uncurry CONJ ((prv cs # prv cs) (dest_conj tm)) ?
    find (\th. aconv (cncl th) tm) cs ?
    COUNTRY tm (find (\th. (cncl th) = "F") cs) in
  let mkasm =
    let swap_eq = mk_eq o (\x,y,z. y,x) o dest_eq in
  let pr1 bdy asm = CONJUNCTS(ASSUME asm) @ CONJUNCTS(ASSUME bdy) in
  let pr2 f bdy asm =
    let eqn = f (ASSUME asm) and cs = CONJUNCTS(ASSUME bdy) in
    CONJUNCTS(EQ_MP eqn (prv cs (lhs (concl eqn)))) @ cs in
  \L r. (let ltm = extr L l in
    (pr2 I, pr2 SYM, mk_eq(ltm, extr L r) ? pr1, pr1, ltm) ?
    (pr1, pr1, extr L r) in
  \(A,g),
  (let l,r = (dest_eq g ? failwith 'goal is not an equivalence') in
  let lcs = conjuncts l and rcs = conjuncts r in
  let cncl = filter (\tm. exists (aconv tm) rcs) lcs in
  if (seq lcs rcs or mem "P" cncl) then
    let imp1 = DISCH l (prv (CONJUNCTS (ASSUME l))) r) and
    imp2 = DISCH r (prv (CONJUNCTS (ASSUME r))) l in
    ACCEPT_TAC (IMP_ANTISYM_RULE imp1 imp2) (A,g) else
  if (null cncl) then failwith 'no conjuncts in common' else
  let p1, p2, asm = mkasm cncl l r in
  \A,asm],
  \[th]. let th1 = (prv (p1 l asm) r) and th2 = (prv (p2 r asm) l) in
  let th1 = PROOF_HYP th th1 and thm2 = PROOF_HYP th th2 in
  let lc = CONJUNCTS(ASSUME l) and rc = CONJUNCTS(ASSUME r) in
  let rth = itlist PROOF_HYP rc thm2 in
  let lth = itlist PROOF_HYP lc thm1 in
  let thm = IMP_ANTISYM_RULE (DISCH l rth) (DISCH r lth) in
  EQ_MP (ALPHA (concl thm) g) thm)
?\st failwith 'CANCEL_CONJ_TAC: ' st;
let EXISTS_ELIM_TAC = CONV_TAC (DEPTH_CONV (CHANGED_CONV (UNWIND_TAC PRUNE)));

let EXISTSF_ELIM_TAC = CONV_TAC (DEPTH_CONV (CHANGED_CONV (UNWINDF_TAC PRUNE)));

let SYM_CONV tm =
  (let (a, b) = dest_eq tm
   in
    let my_CONV = DISCH_ALL o SYM o ASSUME in
    IMP_ANTISYM_RULE (my_CONV "a = b") (my_CONV "b = a")
  )

let SYM_RULE = (CONV_RULE (ONCE_DEPTH_CONV SYM_CONV))

let FORALL_EQ_TAC: tactic(1, t) =
  (let t1, t2 = dest_eq t
    in
     let n1, w1 = dest_forall t1
     and n2, w2 = dest_forall t2
     in
      (n1 = n2)
      =>
        let p[th] = MK_COMB (REFL (fst (dest_comb t1))), ABS n1 th) in
        ([1, mk_eq(w1, w2)], p)
      | fail
    )

let EQ_ANTE_ELIM tm
  = let vars, body = strip_forall tm in
    let x, t = dest_eq (fst (dest_imp body))
    in
    let spec1 = map (\tm. (tm = x) => t | tm) vars
    in
    let genl = filter (\tm. not (tm = x)) vars
    in
    let impl1 = DISCH tm (GENEL genl (MP (SPECL spec1 (ASSUME tm)) (REFL t)))
    in
    let asm1 = SPECL genl (ASSUME (snd (dest_imp (concl impl1)))))
    in
    let th = (DISCH "$x = " t"
      (SUBST [ SYM (ASSUME "$x = " t"), x ]
        (snd (dest_imp body)) asm1))
    in
    let impl2 = DISCH_ALL (GENEL vars th)
    in
    IMP_ANTISYM_RULE impl1 impl2;
Appendix C

Theories developed for this text

C.1 Pairs.ml

new_theory 'pairs';;

let pair_imp_lemma
    = let p = (assume "(a:* b:** = (c, d))")
    in
        disc_all
        (conj
            (rewrite_rule [fst] (ap_term "fst:***" p))
            (rewrite_rule [snd] (ap_term "snd:***" p)))
    ;;

let pair_split = prove_thm
('pair_split',
    "! (a:* b:** c d . ((a, b) = (c, d)) = ((a = c) /
      (b = d))", repeat_gen_tac
    then eq_tac
    thenl
        [ accept_tac pair_imp_lemma
          ;;
        ]);

load_definition 'bool' 'ONE_ONE_DEF';;

let exists_select_lemma = save_thm
('exists_select_lemma',
    select_rule
    (exists
        ("? i . (P:---*) i = P x", "x:*")
        (repl "(P:---*) x")
    ));

let select_11_lemma = prove_thm
('select_11_lemma',
    "! (P:---**) . (one_one P =>> (! x . (0 i . P i = P x) = x))", gen_tac
    then pure_once_rewrite_tac [one_one_def]
    then disc_tac then gen_tac
    then assume_tac exists_select_lemma
    ;;

587
let pair_bool_cases = prove_thm
('pair_bool_cases',
  " ! (x:bool#bool).
    (x = (T, T)) \ (/ (x = (T, F)) \ (/ (x = (F, T)) \ (/ (x = (F, F))"),
GEN_TAC
THEN SUBST1_TAC
  (INST_TYPE [ ("#:bool", ":*"), ("#:bool", ":**") ]
   (SYM(SPEC_ALL PAIR)))
THEN BOOL_CASES_TAC "FST (x:bool#bool)"
THEN REWRITE_TAC [ pair_split; EXCLUDED_MIDDLE ]
);
close_theory();
C.2 Bools.ml

new_theory 'bools';;

let BLATT cases = REPEAT GEN_TAC
   THEN MAP_EVERY BOOL_CASES_TAC cases
   THEN REWRITE_TAC[ ];;

let DISJ_ASSOC = prove_thm
   ('DISJ_ASSOC',
   " ! a b c . (a ∨ (b ∨ c)) = ((a ∨ b) ∨ c)",
   BLATT [ "a:bool" ]);;

let DISJ_DISTRIB1 = prove_thm
   ('DISJ_DISTRIB1',
   " ! a b c . ((a ∨ (b ∧ c)) = (a ∨ b) ∧ (a ∨ c))",
   BLATT [ "a:bool" ]);;

let DISJ_DISTRIB2 = prove_thm
   ('DISJ_DISTRIB2',
   " ! a b c . ((a ∧ (b ∨ c)) = (a ∧ b) ∨ (a ∧ c))",
   BLATT [ "a:bool" ]);;

let CONJ_ASSOC = prove_thm
   ('CONJ_ASSOC',
   " ! a b c . (a ∧ (b ∨ c)) = ((a ∧ b) ∨ (a ∨ c))",
   BLATT [ "a:bool" ]);;

let CONJ_DISTRIB1 = prove_thm
   ('CONJ_DISTRIB1',
   " ! a b c . ((a ∧ (b ∨ c)) = (a ∧ b) ∨ (a ∧ c))",
   BLATT [ "a:bool" ]);;

let CONJ_DISTRIB2 = prove_thm
   ('CONJ_DISTRIB2',
   " ! a b c . ((a ∨ (b ∧ c)) = (a ∨ b) ∨ (a ∨ c))",
   BLATT [ "a:bool" ]);;

let DISJ_DISTRIB = prove_thm
   ('DISJ_DISTRIB',
   "((a∧b∧c) ∧ (a∨b∧c) = (a∧(b∨c)) ∧
   (a∧b) ∧ (a∧c))", ACCEPT_TAC (CONJ DISJ_DISTRIB1 DISJ_DISTRIB2));;

let CONJ_DISTRIB = prove_thm
   ('CONJ_DISTRIB',
   "((a∧b∧c) ∨ (a∨b∧c) = (a∨b) ∨ (a∨c)) ∧
   (a∧b) ∨ (a∧c) ∨ (b∨c) ∨ (b∨c)", ACCEPT_TAC (CONJ CONJ_DISTRIB1 CONJ_DISTRIB2));;

close_theory();
C.3  bits.ml

```ml
new_theory 'bits';
loadf '/home/vlsi/graham/bol-ideas/SYM_CONV';

let bv = new_definition
  ('bv', "! b. bv b = b => SUC 0 | 0");

let vb = new_definition
  ('vb', "! n. vb n = ((n=0) => F | T)");

let bvInverse = prove_thm
  ('bvInverse', "! n. ((n=0) \ (n=1)) => (bv (vb n) = n)",
  GEN_TAC
  THEN STRIP_TAC
  THEN ASM_REWRITE_TAC
  [bv; vb; num_CONV "1"; NOT_SUC];

let vbInverse = prove_thm
  ('vbInverse', "! b. vb (bv b) = b",
  GEN_TAC
  THEN PURE_REWRITE_TAC [bv; vb]
  THEN BOOL_CASES_TAC "b:boolean"
  THEN REWRITE_TAC [NOT_SUC]);

let bvals = prove_thm
  ('bvals', "(bv T = SUC 0) \ (bv F = 0)",
  REWRITE_TAC [bv]);

let ivals = prove_thm
  ('ival', (bv b b = 1) = (bv b = b) /
    ((bv b = SUC 0) = (b = T)) /
    ((bv b = 0) = (b = F)) /
    ((bv b < 1) = (b = F)) /
    ((0 < bv b) = (b = T))",
  GEN_TAC
  THEN BOOL_CASES_TAC "b:boolean"
  THEN REWRITE_TAC
  [bv; num_CONV "1"; NOT_SUC; SYM_RULE NOT_SUC; LESS_O; LESS_REFL]);

let sub_mono_eq = prove
  ("! m n. (SUC n) - (SUC m) = n - m",
  GEN_TAC THEN INDUCT_TAC
  THEN ASM_REWRITE_TAC [SUB; LESS_O; LESS_MONO_EQ]);

let maxbit = prove_thm
  ('maxbit',
  "")
```
" ! b . bv b < 2",
GEN_TAC
THEN PURE_REWRITE_TAC [ bv; num_CONV "2"; num_CONV "1"
] THEN BOOL_CASES_TAC "b:bool"
THEN REWRITE_TAC [ LESS_MONO_EQ; LESS_O ];

let maxbit2 = prove_thm
('maxbit2',
" ! b . bv b \leq 1",
GEN_TAC
THEN PURE_REWRITE_TAC
[ LESS_OR_EQ; bv; num_CONV "2"; num_CONV "1" ]
THEN BOOL_CASES_TAC "b:bool"
THEN REWRITE_TAC
[ LESS_O; LESS_MONO_EQ; INV_SUC_EQ; SYM_RULE NOT_SUC ]
);

let bvFF = prove_thm
('bvFF',
" ! a b . (bv a + (bv b)) = 0) = (\neg a \land \neg b)",
REPEAT GEN_TAC
THEN REWRITE_TAC [ ADD_EQ_0; ivals ];

let bvFT = prove_thm
('bvFT',
" ! a b . (bv a + (bv b)) = 1) = (a = b)",
REPEAT GEN_TAC
THEN MAP_EVERY BOOL_CASES_TAC ['a:bool'; 'b:bool']
THEN REWRITE_TAC
[ bvals; num_CONV "1"; ADD_CLAUSES;
  INV_SUC_EQ; SYM_RULE NOT_SUC; NOT_SUC ]
);

let bvTT = prove_thm
('bvTT',
" ! a b . (bv a + (bv b)) = 2) = (a \land b)",
REPEAT GEN_TAC
THEN MAP_EVERY BOOL_CASES_TAC ['a:bool'; 'b:bool']
THEN REWRITE_TAC
[ bvals; num_CONV "1"; num_CONV "2";
  ADD_CLAUSES; INV_SUC_EQ; SYM_RULE NOT_SUC ]
);

let bvLss = prove_thm
('bvLss',
" ! a b . (bv a < (bv b)) = (\neg a \land b)",
REPEAT GEN_TAC
THEN MAP_EVERY BOOL_CASES_TAC ['a:bool'; 'b:bool']
THEN REWRITE_TAC [ bwvals; LESS_REFL; LESS_O; NOT_LESS_O ]
);

let bvEqI = prove_thm
('bvEqI',
" ! a b . (bv a = (bv b)) = (a = b)",
REPEAT GEN_TAC
THEN MAP_EVERY BOOL_CASES_TAC ['a:bool'; 'b:bool']
THEN REWRITE_TAC
[ bvals; num_CONV "1"; num_CONV "2";
  ADD_CLAUSES; INV_SUC_EQ; SYM_RULE NOT_SUC ]
);
let bvGtr = prove_thm
('bvGtr',
"! a b. (bv a) > (bv b)) = (a \ b)",
  REPEAT GEN_TAC
  THEN PURE_ONCE_REWRITE_TAC [GREATER ]
  THEN PURE_ONCE_REWRITE_TAC [CONJ_SYM]
  THEN REWRITE_TAC [ bvLss ]
);

let bitsLemma = prove_thm
('bitsLemma',
"! a b. (bv a + bv b < 2)) = (bv a + bv b = 2)",
  REPEAT GEN_TAC
  THEN PURE_REWRITE_TAC [num_CONV "2"; num_CONV "1"]
  THEN MAP_EVERY BOOL_CASES_TAC ["a":bool;"b":bool]
  THEN REWRITE_TAC [ bvals ]
);

let bit1Cases = prove_thm
('bit1Cases',
"! a . (bv a = 0) \ (bv a = SUC 0) ",
  GEN_TAC
  THEN BOOL_CASES_TAC "a:bool"
  THEN REWRITE_TAC [ bvals ]
);

let bit2Cases = prove_thm
('bit2Cases',
"! a b . (bv a + bv b = 0) \ (bv a + bv b = SUC 0) \ (bv a + bv b = SUC(SUC 0))",
  REPEAT GEN_TAC
  THEN MAP_EVERY BOOL_CASES_TAC ["a":bool;"b":bool]
  THEN REWRITE_TAC [ bvals; ADD_CLAUSES ]
);

let bit3Cases = prove_thm
('bit3Cases',
"! a b c . (bv a + bv b + bv c = 0) \ (bv a + bv b + bv c = SUC 0) \ (bv a + bv b + bv c = SUC(SUC 0))",
  REPEAT GEN_TAC
  THEN MAP_EVERY BOOL_CASES_TAC ["a":bool;"b":bool;"c":bool]
  THEN REWRITE_TAC [ bvals; ADD_CLAUSES ]
);
C.4  nums.ml

new_theory 'nums';;

map new_parent ['bools'];;
map load_theorems ['bools'];;
loadf '/home/vlsi/graham/hol-ideas/SYM_CONV';;

let SUC_NOT = prove_thm
('SUC_NOT',
"! n . (0 = SUC n)",
ACCEPT_TAC (SYM_RULE NOT_SUC));;

let multby2 = prove_thm
('multby2',
"! n . 2 * n = n + n", GEN_TAC
THEN REWRITE_TAC [num_CONV "2"; MULT_CLAUSES]);;

let mult_eq_0 = prove_thm
('mult_eq_0',
"! a b . (a * b = 0) = ((a = 0) \ (b = 0))", 
INDUCT_TAC THEN GEN_TAC
THEN ASM_REWRITE_TAC [MULT_CLAUSES; ADD_EQ_0 ]
THEN BOOL_CASES_TAC "b = 0"
THEN REWRITE_TAC [NOT_SUC ]);;

let nsubn = prove_thm
('nsubn',
"! n . n - n = 0", GEN_TAC
THEN REWRITE_TAC [SUB_EQ_0; LESS_OR_EQ ]);;

let sub_mono_eq = prove_thm
('sub_mono_eq',
"! m n . (SUC n) - (SUC m) = n - m", 
GEN_TAC THEN INDUCT_TAC
THEN ASM_REWRITE_TAC [SUB; LESS_0; LESS_MONO_EQ ]);;

let sub_same_eq = prove_thm
('sub_same_eq',
"! a b c . ((b + a) - (c + a)) = (b - c)", 
INDUCT_TAC
THEN REPEAT GEN_TAC
THEN ASM_REWRITE_TAC [ADD_CLAUSES; sub_mono_eq ]];;

let lss_sub_0 = prove_thm
('lss_sub_0',
"! m n . m <= n => (m - n = 0)",
REWRITE_TAC [ snd(EQ_IMP_RULE SPEC_ALL SUB_EQ_0) ]);;

let sub_lss_0 = prove_thm
let add_gtr_sub = prove_thm
('add_gtr_sub',
  "!a b c. (\(c = 0\) /\ \(c <= a\)) \rightarrow (a + b = (a - c))",
REPEAT GEN_TAC THEN STRIP_TAC
THEN IMP_RES_TAC SUB_ADD
THEN ASM_REWRITE_TAC
[ SYM (SPECL ['"a+b"'; '"a-c"'; '"c:num"'] EQ_MONO_ADD_EQ);
  SYM_RULE ADD_ASSOC;
  ADD_INV_EQ; ADD_EQ_O; DE_MORGAN_THM ]);;

let lss_mono = prove_thm
('lss_mono',
  "!a . a < SUC a",
INDUCT_TAC
THEN ASM_REWRITE_TAC [ LESS_O; LESS_MONO_EQ ]);;

let lss_lss_add = prove_thm
('lss_lss_add',
  "!a b x. (x < a) \rightarrow x < (a + b)",
GEN_TAC THEN INDUCT_TAC THEN GEN_TAC
THEN
[ REWRITE_TAC [ ADD_CLAUSES ]
;
  REWRITE_TAC [ ADD_CLAUSES ]
  THEN STRIP_TAC THEN RES_TAC
  THEN ACCEPT_TAC
  (MATCH_MP LESS_TRANS
   (COND (ASSUME "x < (a + b)"
     (SPEC "a+b" lss_mono)))];
]

let lss_lss_add_lss = prove_thm
('lss_lss_add_lss',
  "!a b c d. (a < b) /\ (c < d) \rightarrow ((a + c) < (b + d))",
REPEAT GEN_TAC
THEN let th = SYM_RULE LESS_MONO_ADD_EQ in
SUBST_TAC
[ SPECL ['"a:num"'; '"b:num"'; '"c:num"'] th;
  ONCE_REWRITE_RULE [ ADD_SYM ]
  (SPECL ['"c:num"'; '"d:num"'; '"b:num"'] th)
]
THEN REWRITE_TAC [ LESS_TRANS ]);

let lss_leq_add_lss = prove_thm
('lss_leq_add_lss',
  "!a b c d . (a < b) /\ (c <= d) \rightarrow ((a + c) < (b + d))"
let lss_leq_trims = prove_thm
  ('lss_leq_trims','
  " ! n m p. (n < m) \& (m <= p) ==> (n < p)",
  REPEAT GEN_TAC
  THEN PURE_ONCE_REWRITE_TAC [LESS_OR_EQ ]
  THEN REPEAT STRIP_TAC
  THEN [ IMP_RES_TAC lss_lss_add_lss
    ; ASM_REWRITE_TAC [LESS_MONO_ADD_EQ ]
  ];);

let leq_lss_trims = prove_thm
  ('leq_lss_trims','
  " ! n m p. (n <= m) \& (m < p) ==> (n < p)",
  REPEAT GEN_TAC
  THEN PURE_ONCE_REWRITE_TAC [LESS_OR_EQ ]
  THEN REPEAT STRIP_TAC
  THEN [ IMP_RES_TAC LESS_TRANS
    ; PURE_ONCE_REWRITE_TAC [SYM_RULE(ASSUME "m = (p:num)")]
    THEN ASM_REWRITE_TAC []
  ];);

let add_sub = prove_thm
  ('add_sub','
  " ! n m . (n + m) - m = n",
  GEN_TAC THEN INDUCT_TAC
  THEN ASM_REWRITE_TAC [ADD_CLAUSES; SUB_0; sub_mono_eq ]);

let sub_add_assoc = prove_thm
  ('sub_add_assoc','
  " ! a b c . ((a - b) - c) = (a - (b + c))",
  INDUCT_TAC THEN INDUCT_TAC THEN GEN_TAC
  THEN ASM_REWRITE_TAC [ADD_CLAUSES; SUB_0; sub_mono_eq ]);

let contra = prove_thm
  ('contra',
  " ! p q . (p => q) = (\~ q => \~ p)",
  REPEAT GEN_TAC
  THEN BOOL_CASES_TAC "p:bool"
  THEN REWRITE_TAC []);
let lss_not_rev = prove_thm
  ("lss_not_rev`,
   " ! a b . a < b ==> ~(b < a)`,
   REPEAT INDUCT_TAC
   THEN ASM_REWRITE_TAC [LESS_REFL; NOT_LESS_O; LESS_MONO_EQ]);;

let lss_not_add_lss = prove_thm
  ("lss_not_add_lss`,
   " ! a b c . (c < b) ==> ~(a+b) < c`,
   REPEAT INDUCT_TAC
   THEN REWRITE_TAC [ADD_CLAUSES; LESS_REFL; NOT_LESS_O; LESS_MONO_EQ]
   THEN STRIP_TAC
   THENL
   [ ACCEPT_TAC (REWRITE_RULE [ADD_CLAUSES] (ASSUME "~(0+b) < c")
   ; ASM_REWRITE_TAC [SYM_RULE (e1 3 (CONJUNCTS ADD_CLAUSES))]);
   ];

let not_add_lss = prove_thm
  ("not_add_lss`,
   " ! a b . ~(a+b) < b`,
   PURE_ONCE_REWRITE_TAC [NOT_LESS]
   THEN PURE_ONCE_REWRITE_TAC [ADD_SYM]
   THEN REWRITE_TAC [LESS_EQ_ADD];;

let add_sub_assoc = prove_thm
  ("add_sub_assoc`,
   " ! a b c . c <= b ==> (a + b - c = a + (b - c))`,
   PURE_REWRITE_TAC [LESS_OR_EQ]
   THEN REPEAT INDUCT_TAC
   THEN ASM_REWRITE_TAC [SUB_O; ADD_CLAUSES]
   THENL
   [ REWRITE_TAC [NOT_LESS_O; NOT_SUC]
   ;
   PURE_REWRITE_TAC
   [LESS_MONO_EQ; INV_SUC_EQ; sub_mono_eq]
   THEN ASM_REWRITE_TAC
   [SYM_RULE (e1 3 (CONJUNCTS ADD_CLAUSES))];
   ];

let x1 = TAC_PROOF
  (([`0 = m`],
   GEN_TAC THEN REWRITE_TAC [SUB_O]);

let x2 = TAC_PROOF
  (([`m = 0`],
   GEN_TAC THEN REWRITE_TAC [SUB_O]);

let x3 = TAC_PROOF
  (([`m = 0`],
   " ! m . (n = 0) ==> (m - n) < (m - (PRE m))`),
   REPEAT GEN_TAC THEN STRIP_TAC
   THEN STRIP_ASSUME_TAC (SPEC "n:num" num_CASES)
   THENL
   [ RES_TAC]
let x4 = TAC_PROOF
  (([], "! m n . m - (SUC n) = PRE (m - n)")
  REPEAT GEN_TAC
  THEN STRIP_ASSUME_TAC (SPEC "m:num" num_CASES)
  THENL
  [ ASM_REWRITE_TAC [ SUB_mono_eq ]
  THENL
    [ ASM_REWRITE_TAC [ SUB_mono_eq; SUB ]
    THEN ASM_CASES_TAC "n' < n"
    THEN ASM_REWRITE_TAC [ ]
    THENL
      [ IMP_RES_TAC (REWRITE_RULE [ LESS_OR_EQ ] lss_sub_0)
      THEN ASM_REWRITE_TAC [ PRE ]
      ]
  ]
  ]
);;

let SUB_CLAUSES = save_thm
  ("SUB_CLAUSES", (CONJ x1 (CONJ x2 (CONJ x3 x4))));;

close_theory();;
C.5 words.ml

new_theory 'words';

let L = ['bits'; 'nums'];
map new_parent L;;
map load_theorems L;;
map load_definitions L;;

loadf '/home/vlsi/silind/HOL2/library/unwind/des-unwind';
loadf '/home/vlsi/graham/hol-ideas/EXTRAS';
loadf '/home/vlsi/graham/hol-ideas/SYM_CONV';
loadf '/home/vlsi/graham/hol-ideas/CANCEL_CONV1_TAC';

let exp_pos = prove_thm
('exp_pos',
"! n . 0 < (2 EXP n)",
INDUCT_TAC
THEN POP_ASM_REWRITE_TAC
[ EXP; num_CONV "1"; LESS_0; multby2 ]
THEN ASSUM_TAC (SPECL ["2 EXP n"; "2 EXP n"] LESS_EQ_ADD)
THEN IMP_RES_TAC less_leq_trans);

let exp_not_0 = prove_thm
('exp_not_0',
"! n . ~(2 EXP n = 0)",
GEN_TAC
THEN ACCEPT_TAC
(SYM_RULE
(MATCH_MP LESS_NOT_EQ (SPEC_ALL exp_pos))));

let exp_mono = prove_thm
('exp_mono',
"! n . (2 EXP n) < (2 EXP (SUC n))",
GEN_TAC
THEN
let lem = REWRITE_RULE [ ADD_CLAUSES ]
(SPEC "0" LESS_MONO_ADD_EQ)
in
REWRITE_TAC [ EXP; multby2; lem; exp_pos ]);;

let exp_doubles = prove_thm
('exp_doubles',
"! n . (2 EXP (SUC n)) = (2 EXP n) + (2 EXP n)",
REWRITE_TAC [ EXP; multby2 ]);;

let val = new_prim_rec_definition
('val',
"(val f 0 = bv (f 0)) /
(val f (SUC n) = val f n + ((2 EXP (SUC n)) * (bv (f (SUC n)))))
");;

let maxword = prove_thm
let maxword2 = prove_thm
('maxword2',
"! n a c i n . ((val a n) < (2 EXP (SUC n)))",
REPEAT GEN_TAC
THEN PURE_ONCE_REWRITE_TAC [ val ]
THENL
[ PURE_REWRITE_TAC [ EXP; MULT_CLAUSES; maxbit ]
; BOOL_CASES_TAC "a(SUC n):bool"
THEN REWRITE_TAC [ bv; ADD_CLAUSES; MULT_CLAUSES ]
THENL
[ PURE_ASM_REWRITE_TAC
 [ SPEC "SUC n" exp_doubles; LESS_MONO_ADD_EQ ]
; ACCEPT_TAC
 (MATCH_MP LESS_TRANS
 (CONJ
 (SPEC_ALL (ASSUME "! n a . (val a n) < (2 EXP (SUC n)))")
 (SPEC "SUC n" exp_mono))
)]]
);

let abcLessDblExp = prove_thm
('abcLessDblExp',
"! n a b c i n . (val a n + val b n + bv c i n) < (2 EXP (SUC(SUC n)))",
REPEAT GEN_TAC
THEN PURE_ONCE_REWRITE_TAC [ exp_doubles ]
THEN ACCEPT_TAC
 (MATCH_MP less_leq_add_less
 (CONJ (SPEC_ALL maxword)
 (SPECL [ "n:num"; "b:num->bool"; "cin:bool" ] maxword2)))]
);

let nLss = new_prim_rec_definition
('nLss',
"(nLss a b 0 = (bv(a 0) < bv(b 0)))
\(nLss a b (SUC n) = (bv(a(SUC n)) < bv(b(SUC n)))
\/ ((nLss a b n) \& (bv(a(SUC n)) = bv(b(SUC n))))\)"
let nEq1 = new_prim_rec_definition
  ('nEq1',
  "(nEq1 a (num->bool) b 0 = (bv a 0) = bv b 0))
  /
  (nEq1 a b (SUC n) = (nEq1 a b n)
     /
     (bv a (SUC n)) = bv (b (SUC n))))
  ");

let nGtr = new_prim_rec_definition
  ('nGtr',
  "(nGtr a b 0 = (bv a 0) > bv b 0))
  /
  (nGtr a b (SUC n) = (bv a (SUC n)) > bv b (SUC n))
  "/ (nGtr a b n) /
  (bv a (SUC n)) = bv b (SUC n))))
  ");

let nLssVal = prove_thm
  ('nLssVal',
  " l n a b . (nLss a b n) = (val a n < val b n)",
  INDUCT_TAC THEN REPEAT GEN_TAC
  THEN ASM_REWRITE_TAC [ nLss; val ]
  THEN MAP_EVERY BOOL_CASES_TAC
  [ "(a (SUC n)):bool"; "(b (SUC n)):bool" ]
  THEN PURE_ONCE_REWRITE_TAC [ bvLss; bvEq ]
  THEN REWRITE_TAC [ bvLss; MULT_CLAUSES; ADD_CLAUSES ]
  THENL
  let lem1 = MATCH_MP lss_lss_add (SPEC_ALL maxword) in
  let lem2 = MATCH_MP LESS_NOT_EQ lem1 in
  let lem3 = MATCH_MP lss_not_rev lem2 in
  let lem4 = MATCH_ACCEPT_TAC lem3 in
  MATCH_ACCEPT_TAC lem4

let nEq1Val = prove_thm
  ('nEq1Val',
  " l n a b . (nEq1 a b n) = (val a n = val b n)",
  INDUCT_TAC THEN REPEAT GEN_TAC
  THEN ASM_REWRITE_TAC [ nEq1; val; bvEq ]
  THEN MAP_EVERY BOOL_CASES_TAC
  [ "(a (SUC n)):bool"; "(b (SUC n)):bool" ]
  THEN REWRITE_TAC [ bvLss; val; MULT_CLAUSES; ADD_CLAUSES ]
  THENL
  let lem1 = MATCH_MP lss_lss_add (SPEC_ALL maxword) in
  let lem2 = MATCH_MP LESS_NOT_EQ lem1 in
  let lem3 = MATCH_MP lss_not_rev lem2 in
  let lem4 = MATCH_ACCEPT_TAC lem3 in
  MATCH_ACCEPT_TAC lem4


let nGtrVal = prove_thm
('nGtrVal',
"! n a b. nGtr a b n = (val a n) > (val b n)",
INDUCT_TAC THEN REPEAT GEN_TAC
THEN ASM_REWRITE_TAC
[ nGtr; GREATER; SYM_RULE nLssVal; nLss ]
THEN SUBSTIT_TAC
(SPECL
[ "bv(a(SUC n))"; "bv(b(SUC n))" ]
(INST_TYPE [ ("*:num", "*:" ) ] EQ_SYM_EQ))
THEN REFL_TAC);

let nNot = new_definition
('nNot', "nNot (a: num -> bool) n = (~ (a n))");

let nAnd = new_definition
('nAnd', "nAnd (a: num -> bool) b n = (a n /\ b n)");

let nOr = new_definition
('nOr', "nOr (a: num -> bool) b n = (a n /\ b n)");

let nXor = new_definition
('nXor', "nXor (a: num -> bool) b n = ~(a n = b n)");

let nNOT = new_definition
('nNOT',
"nNOT n a s = (nEql s (nNot a) n)");

let nAND = new_definition
('nAND',
"nAND n a b s = (nEql s (nAnd a b) n)");

let nOR = new_definition
('nOR',
"nOR n a b s = (nEql s (nOr a b) n)");

let nXOR = new_definition
('nXOR',
"nXOR n a b s = (nEql s (nXor a b) n)");

let allFalse = new_definition
('allFalse', "! (n: num) . allFalse n = F");

let allTrue = new_definition
('allTrue', "! (n: num) . allTrue n = T");

let valAllFalse = prove_thm
('valAllFalse',
"! n. valAllFalse n = 0",
INDUCT_TAC
THEN ASM_REWRITE_TAC
let valAllTrue = prove_thm
('valAllTrue',
" ! n . val allTrue n = (2 EXP (SUC n)) - (SUC 0)",
INDUCT_TAC
THEN PURE_ASM_REWRITE_TAC [ allTrue; val; bvals ]
THEN
[ PURE_REWRITE_TAC [ EXP; ADD_CLAUSES; MULT_CLAUSES ]
THEN REWRITE_TAC
[ num_CONV "2"; num_CONV "1"; sub_mono_eq; SUB_0 ]
;
PURE_REWRITE_TAC
[ ADD_CLAUSES; MULT_CLAUSES; SPEC "SUC n" exp_doubles ]
THEN
  let lem1 = MATCH_MP LESS_OR (SPEC_ALL exp_pos) in
  let lem2 = MATCH_MP add_sub_assoc lem1 in
  PURE_ONCE_REWRITE_TAC [ lem2 ]
  THEN MATCH_ACCEPT_TAC ADD_SYM
)];);

let valSucAllTrue = prove_thm
('valSucAllTrue',
" ! n . SUC(val allTrue n) = (2 EXP (SUC n))", 
INDUCT_TAC
THEN PURE_REWRITE_TAC [ val; allTrue; bvals; MULT_CLAUSES ]
THEN
[ REWRITE_TAC
  [ EXP; MULT_CLAUSES; num_CONV "2"; num_CONV "1"; ADD_CLAUSES ]
;
PURE_ONCE_REWRITE_TAC
[ SPEC "SUC n" exp_doubles; el 2 (CONJUNCTS ADD_CLAUSES) ]
THEN PURE_ONCE_REWRITE_TAC
[ SYM_RULE (el 3 (CONJUNCTS ADD_CLAUSES)) ]
THEN ASM_REWRITE_TAC [ EQ_MONO_ADD_EQ ]
)];);

let bAddNotb = prove_thm
('bAddNotb',
" ! n b . (val b n) + (val (n\not b) n) = val allTrue n", 
INDUCT_TAC THEN GEN_TAC
THEN REWRITE_TAC [ val; n\not; allTrue ]
THEN
[ BOOL_CASES_TAC "b 0:bool"
  THEN REWRITE_TAC [ bvals; ADD_CLAUSES ]
;
BOOL_CASES_TAC "b(SUC n):bool"
  THEN REWRITE_TAC [ bvals; ADD_CLAUSES; MULT_CLAUSES ]
THEN
[ PURE_ONCE_REWRITE_TAC [ SYM_RULE ADD_ASSOC ]
  THEN PURE_ONCE_REWRITE_TAC [ SPEC "2 EXP (SUC n)" ADD_SYM ]
;
ALL_TAC]
let valNot = prove_thm
  (valNot',
   "\! a b. (val (\not b) n) = (2 EXP (Suc n)) - Suc(val b n)",
   REPEAT GEN_TAC
   THEN
     let lem1 = SPECL ["val (\not b) n"; "(2 EXP (Suc n)) - Suc(val b n)"; "Suc(val b n)"
       ] EQ_MONO_ADD_EQ
     in
     let lem2 = PURE_ONCE_REWRITE_RULE [LESS_EQ] maxword in
     let lem3 = MATCH_MP SUB_ADD
       (SPECL ["n:num"; "b:num->bool"] lem2) in
     MAP_EVERY SUBST_TAC [SYM_RULE lem1; lem3 ]
     THEN PURE_ONCE_REWRITE_TAC [ADD_CLAUSES]
     THEN PURE_ONCE_REWRITE_TAC
       [PURE_ONCE_REWRITE_RULE [ADD_SYM] bAddNotb ]
     THEN MATCH_ACCEPT_TAC valSucAllTrue);;

close_theory();;
References


93/626, volume 1, US Department of Commerce, National Institute of Standards and Technology, Gaithersburg MD, 1993.


[54] F. Hennie. 


47, Computer Science Department, University of Austin at Texas, Austin, Texas, 1985.


Preface


