The Isabelle/Isar Reference Manual

Makarius Wenzel

With Contributions by Clemens Ballarin, Stefan Berghofer, Jasmin Blanchette, Timothy Bourke, Lukas Bulwahn, Lucas Dixon, Florian Haftmann, Brian Huffman, Gerwin Klein, Alexander Krauss, Ondřej Kunčar, Tobias Nipkow, Lars Noschinski, David von Oheimb, Larry Paulson, Sebastian Skalberg

May 22, 2012
The Isabelle system essentially provides a generic infrastructure for building deductive systems (programmed in Standard ML), with a special focus on interactive theorem proving in higher-order logics. Many years ago, even end-users would refer to certain ML functions (goal commands, tactics, tacticals etc.) to pursue their everyday theorem proving tasks.

In contrast Isar provides an interpreted language environment of its own, which has been specifically tailored for the needs of theory and proof development. Compared to raw ML, the Isabelle/Isar top-level provides a more robust and comfortable development platform, with proper support for theory development graphs, managed transactions with unlimited undo etc. The Isabelle/Isar version of the Proof General user interface [2, 3] provides a decent front-end for interactive theory and proof development in this advanced theorem proving environment, even though it is somewhat biased towards old-style proof scripts.

Apart from the technical advances over bare-bones ML programming, the main purpose of the Isar language is to provide a conceptually different view on machine-checked proofs [45, 46]. Isar stands for Intelligible semi-automated reasoning. Drawing from both the traditions of informal mathematical proof texts and high-level programming languages, Isar offers a versatile environment for structured formal proof documents. Thus properly written Isar proofs become accessible to a broader audience than unstructured tactic scripts (which typically only provide operational information for the machine). Writing human-readable proof texts certainly requires some additional efforts by the writer to achieve a good presentation, both of formal and informal parts of the text. On the other hand, human-readable formal texts gain some value in their own right, independently of the mechanic proof-checking process.

Despite its grand design of structured proof texts, Isar is able to assimilate the old tactical style as an “improper” sub-language. This provides an easy upgrade path for existing tactic scripts, as well as some means for interactive experimentation and debugging of structured proofs. Isabelle/Isar supports a broad range of proof styles, both readable and unreadable ones.

The generic Isabelle/Isar framework (see chapter 2) works reasonably well
for any Isabelle object-logic that conforms to the natural deduction view of the Isabelle/Pure framework. Specific language elements introduced by the major object-logics are described in chapter 10 (Isabelle/HOL), chapter 11 (Isabelle/HOLCF), and chapter 12 (Isabelle/ZF). The main language elements are already provided by the Isabelle/Pure framework. Nevertheless, examples given in the generic parts will usually refer to Isabelle/HOL as well.

Isar commands may be either *proper* document constructors, or *improper commands*. Some proof methods and attributes introduced later are classified as improper as well. Improper Isar language elements, which are marked by “∗” in the subsequent chapters; they are often helpful when developing proof documents, but their use is discouraged for the final human-readable outcome. Typical examples are diagnostic commands that print terms or theorems according to the current context; other commands emulate old-style tactical theorem proving.
## Contents

I Basic Concepts

1 Synopsis

1.1 Notepad

1.1.1 Types and terms

1.1.2 Facts

1.1.3 Block structure

1.2 Calculational reasoning

1.2.1 Special names in Isar proofs

1.2.2 Transitive chains

1.2.3 Degenerate calculations and bigstep reasoning

1.3 Induction

1.3.1 Induction as Natural Deduction

1.3.2 Induction with local parameters and premises

1.3.3 Implicit induction context

1.3.4 Advanced induction with term definitions

1.4 Natural Deduction

1.4.1 Rule statements

1.4.2 Isar context elements

1.4.3 Pure rule composition

1.4.4 Structured backward reasoning

1.4.5 Structured rule application

1.4.6 Example: predicate logic

1.5 Generalized elimination and cases

1.5.1 General elimination rules

1.5.2 Rules with cases

1.5.3 Obtaining local contexts

2 The Isabelle/Isar Framework
2.1 The Pure framework ........................................ 27
  2.1.1 Primitive inferences .................................. 28
  2.1.2 Reasoning with rules .................................. 29
2.2 The Isar proof language ................................. 31
  2.2.1 Context elements ..................................... 32
  2.2.2 Structured statements ................................ 34
  2.2.3 Structured proof refinement ......................... 35
  2.2.4 Calculational reasoning .............................. 37
2.3 Example: First-Order Logic ............................. 38
  2.3.1 Equational reasoning ................................. 39
  2.3.2 Basic group theory .................................... 40
  2.3.3 Propositional logic .................................. 41
  2.3.4 Classical logic ...................................... 43
  2.3.5 Quantifiers .......................................... 44
  2.3.6 Canonical reasoning patterns ....................... 45

II General Language Elements ................................. 48

3 Outer syntax — the theory language ................... 49
  3.1 Lexical matters ........................................... 50
  3.2 Common syntax entities .................................. 52
    3.2.1 Names .............................................. 52
    3.2.2 Numbers .......................................... 53
    3.2.3 Comments ......................................... 53
    3.2.4 Type classes, sorts and arities .................... 54
    3.2.5 Types and terms ................................... 54
    3.2.6 Term patterns and declarations ................... 56
    3.2.7 Attributes and theorems ............................ 57

4 Document preparation ...................................... 61
  4.1 Markup commands ....................................... 62
  4.2 Document Antiquotations ............................... 64
    4.2.1 Styled antiquotations ................................ 68
    4.2.2 General options .................................... 69
  4.3 Markup via command tags ................................ 70
CONTENTS

4.4 Railroad diagrams ........................................... 71
4.5 Draft presentation ........................................... 75

5 Specifications ................................................. 76
5.1 Defining theories ........................................... 76
5.2 Local theory targets ......................................... 78
5.3 Bundled declarations ......................................... 80
5.4 Basic specification elements ................................. 81
5.5 Generic declarations ......................................... 84
5.6 Locales ....................................................... 85
   5.6.1 Locale expressions ...................................... 85
   5.6.2 Locale declarations ...................................... 86
   5.6.3 Locale interpretations ................................... 89
5.7 Classes ....................................................... 92
   5.7.1 The class target ......................................... 95
   5.7.2 Co-regularity of type classes and arities .............. 95
5.8 Unrestricted overloading ...................................... 96
5.9 Incorporating ML code ....................................... 97
5.10 Primitive specification elements ............................ 99
   5.10.1 Type classes and sorts .................................. 99
   5.10.2 Types and type abbreviations ........................... 100
   5.10.3 Constants and definitions ............................... 101
5.11 Axioms and theorems ....................................... 102
5.12 Oracles ....................................................... 103
5.13 Name spaces ................................................ 104

6 Proofs .......................................................... 106
6.1 Proof structure .............................................. 106
   6.1.1 Formal notepad .......................................... 106
   6.1.2 Blocks .................................................... 107
   6.1.3 Omitting proofs .......................................... 108
6.2 Statements ................................................... 108
   6.2.1 Context elements ........................................ 108
   6.2.2 Term abbreviations ...................................... 110
   6.2.3 Facts and forward chaining ............................. 111
## CONTENTS

8.1 Inspecting the context ........................................ 163  
8.2 System commands ............................................. 166  

9 Generic tools and packages ..................................... 167  
9.1 Configuration options .......................................... 167  
9.2 Basic proof tools .............................................. 168  
  9.2.1 Miscellaneous methods and attributes ..................... 168  
  9.2.2 Low-level equational reasoning ......................... 171  
  9.2.3 Further tactic emulations ............................... 173  
9.3 The Simplifier .................................................. 176  
  9.3.1 Simplification methods ................................... 176  
  9.3.2 Declaring rules ........................................... 178  
  9.3.3 Congruence rules ......................................... 178  
  9.3.4 Simplification procedures .............................. 180  
  9.3.5 Forward simplification .................................. 181  
9.4 The Classical Reasoner ....................................... 182  
  9.4.1 Basic concepts ........................................... 182  
  9.4.2 Rule declarations ......................................... 186  
  9.4.3 Structured methods ...................................... 188  
  9.4.4 Automated methods ..................................... 189  
  9.4.5 Semi-automated methods ............................... 193  
  9.4.6 Single-step tactics ..................................... 194  
9.5 Object-logic setup ............................................. 195  

III Object-Logics .................................................. 197  

10 Isabelle/HOL ..................................................... 198  
10.1 Higher-Order Logic ........................................... 198  
10.2 Inductive and coinductive definitions ...................... 199  
  10.2.1 Derived rules ........................................... 201  
  10.2.2 Monotonicity theorems ................................ 202  
10.3 Recursive functions .......................................... 203  
  10.3.1 Proof methods related to recursive definitions ....... 208  
  10.3.2 Functions with explicit partiality .................... 210  
  10.3.3 Old-style recursive function definitions (TFL) .... 211
CONTENTS

10.4 Datatypes .................................................. 213
10.5 Records .................................................. 214
  10.5.1 Basic concepts ...................................... 215
  10.5.2 Record specifications ............................... 216
  10.5.3 Record operations ................................. 217
  10.5.4 Derived rules and proof tools ..................... 218
10.6 Adhoc tuples ............................................. 219
10.7 Typedef axiomatization .................................. 219
10.8 Functorial structure of types ........................... 222
10.9 Transfer package ......................................... 223
10.10 Lifting package ......................................... 224
10.11 Quotient types .......................................... 226
10.12 Coercive subtyping ..................................... 229
10.13 Arithmetic proof support .............................. 230
10.14 Intuitionistic proof search ............................. 230
10.15 Model Elimination and Resolution ..................... 231
10.16 Coherent Logic .......................................... 232
10.17 Proving propositions .................................... 232
10.18 Checking and refuting propositions ................... 234
10.19 Unstructured case analysis and induction ............ 238
10.20 Executable code ........................................ 240
10.21 Definition by specification ........................... 250

11 Isabelle/HOLCF .............................................. 252
  11.1 Mixfix syntax for continuous operations .............. 252
  11.2 Recursive domains ..................................... 252

12 Isabelle/ZF .................................................. 254
  12.1 Type checking ........................................... 254
  12.2 (Co)Inductive sets and datatypes ..................... 254
    12.2.1 Set definitions ................................... 254
    12.2.2 Primitive recursive functions .................... 257
    12.2.3 Cases and induction: emulating tactic scripts ... 257
IV Appendix

A Isabelle/Isar quick reference

A.1 Proof commands ............................................. 260
A.1.1 Primitives and basic syntax .......................... 260
A.1.2 Abbreviations and synonyms .......................... 261
A.1.3 Derived elements ....................................... 261
A.1.4 Diagnostic commands ................................... 261
A.2 Proof methods .............................................. 262
A.3 Attributes ..................................................... 263
A.4 Rule declarations and methods .......................... 263
A.5 Emulating tactic scripts ................................... 264
A.5.1 Commands ................................................ 264
A.5.2 Methods .................................................. 264

B Predefined Isabelle symbols

C ML tactic expressions

C.1 Resolution tactics .......................................... 271
C.2 Simplifier tactics .......................................... 272
C.3 Classical Reasoner tactics ............................... 272
C.4 Miscellaneous tactics ...................................... 273
C.5 Tacticals ..................................................... 273
Part I

Basic Concepts
Chapter 1

Synopsis

1.1 Notepad

An Isar proof body serves as mathematical notepad to compose logical content, consisting of types, terms, facts.

1.1.1 Types and terms

def notepad begin

Locally fixed entities:

fix x — local constant, without any type information yet
fix x :: 'a — variant with explicit type-constraint for subsequent use

fix a b
assume a = b — type assignment at first occurrence in concrete term

Definitions (non-polymorphic):
def x ≡ t::'a

Abbreviations (polymorphic):
let ?f = λx. x
term ?f ?f

Notation:
write x (***)
end

1.1.2 Facts

A fact is a simultaneous list of theorems.
Producing facts

notepad
begin

Via assumption ("lambda"):
  assume $a : A$

Via proof ("let"):
  have $b : B$ sorry

Via abbreviation ("let"):
  note $c = a \ b$

end

Referencing facts

notepad
begin

Via explicit name:
  assume $a : A$
  note $a$

Via implicit name:
  assume $A$
  note this

Via literal proposition (unification with results from the proof text):
  assume $A$
  note "A"

  assume $\forall x. B \ x$
  note "B a" 
  note "B b" 

end

Manipulating facts

notepad
begin

Instantiation:
CHAPTER 1. SYNOPSIS

\textbf{assume} \( a : \bigwedge x. B x \)
\textbf{note} \( a \)
\textbf{note} \( a \ [\text{of } b] \)
\textbf{note} \( a \ [\text{where } x = b] \)

Backchaining:
\begin{itemize}
  \item \textbf{assume} 1: \( A \)
  \item \textbf{assume} 2: \( A \implies C \)
  \item \textbf{note} 2 \([OF 1]\)
  \item \textbf{note} 1 \([THEN 2]\)
\end{itemize}

Symmetric results:
\begin{itemize}
  \item \textbf{assume} \( x = y \)
  \item \textbf{note this} \([\text{symmetric}]\)
  \item \textbf{assume} \( x \neq y \)
  \item \textbf{note this} \([\text{symmetric}]\)
\end{itemize}

Adhoc-simplification (take care!):
\begin{itemize}
  \item \textbf{assume} \( P ([] @ xs) \)
  \item \textbf{note this} \([\text{simplified}]\)
\end{itemize}

\textbf{end}

\section*{Projections}

Isar facts consist of multiple theorems. There is notation to project interval ranges.

\textbf{notepad}
\textbf{begin}
\begin{itemize}
  \item \textbf{assume} \textit{stuff}: \( A B C D \)
  \item \textbf{note} \textit{stuff}(1)
  \item \textbf{note} \textit{stuff}(2−3)
  \item \textbf{note} \textit{stuff}(2−)
\end{itemize}
\textbf{end}

\section*{Naming conventions}

\begin{itemize}
  \item Lower-case identifiers are usually preferred.
  \item Facts can be named after the main term within the proposition.
  \item Facts should \textit{not} be named after the command that introduced them (\textbf{assume, have}). This is misleading and hard to maintain.
\end{itemize}
• Natural numbers can be used as “meaningless” names (more appropriate than $a_1$, $a_2$ etc.)

• Symbolic identifiers are supported (e.g. *, **, ***).

1.1.3 Block structure

The formal notepad is block structured. The fact produced by the last entry of a block is exported into the outer context.

```
notepad
begin
{
  have a: A sorry
  have b: B sorry
  note a b
}
note this
note 'A'
note 'B'
end
```

Explicit blocks as well as implicit blocks of nested goal statements (e.g. `have`) automatically introduce one extra pair of parentheses in reserve. The `next` command allows to “jump” between these sub-blocks.

```
notepad
begin

{
  have a: A sorry
next
  have b: B
  proof –
    show B sorry
next
  have c: C sorry
next
  have d: D sorry
qed
}
```

Alternative version with explicit parentheses everywhere:

```
notepad
begin

{
  have a: A sorry
next
  have b: B
  proof –
    show B sorry
next
  have c: C sorry
next
  have d: D sorry
qed
}
```


d{have a: A sorry}
d{have b: B proof –
  { show B sorry
  }
  { have c: C sorry
  }
  { have d: D sorry
  } qed
}
end

1.2 Calculational reasoning

For example, see "/src/HOL/Isar_Examples/Group.thy.

1.2.1 Special names in Isar proofs

- term \texttt{thesis} — the main conclusion of the innermost pending claim
- term \ldots — the argument of the last explicitly stated result (for infix application this is the right-hand side)
- fact \texttt{this} — the last result produced in the text

notepad
begin
  have x = y
  proof –
  term \texttt{thesis}
  show \texttt{thesis sorry}
  term \texttt{thesis} — static!
CHAPTER 1. SYNOPSIS

Calculational reasoning maintains the special fact called “calculation” in the background. Certain language elements combine primary this with secondary calculation.

1.2.2 Transitive chains

The Idea is to combine this and calculation via typical trans rules (see also print_trans_rules):

```
thm trans
thm less_trans
thm less_le_trans
```

Plain bottom-up calculation:

```
have a = b sorry
also
have b = c sorry
also
have c = d sorry
finally
have a = d .
```

Variant using the ... abbreviation:

```
have a = b sorry
also
have ... = c sorry
also
have ... = d sorry
finally
have a = d .
```

Top-down version with explicit claim at the head:

```
have a = d
proof –
    have a = b sorry
```
also
have \ldots = c \text{ sorry}
also
have \ldots = d \text{ sorry}
finally
\text{show } ?\text{thesis} .
\text{qed}
next

Mixed inequalities (require suitable base type):
\begin{verbatim}
fix a b c d :: nat
have a < b \text{ sorry}
also
have b \leq c \text{ sorry}
also
have c = d \text{ sorry}
finally
have a < d .
end
\end{verbatim}

Notes
\begin{itemize}
\item The notion of \textit{trans} rule is very general due to the flexibility of Isabelle/Pure rule composition.
\item User applications may declare their own rules, with some care about the operational details of higher-order unification.
\end{itemize}

1.2.3 Degenerate calculations and bigstep reasoning

The Idea is to append \textit{this} to \textit{calculation}, without rule composition.

\begin{verbatim}
notepad
begin
A vacuous proof:
have A \text{ sorry}
moreover
have B \text{ sorry}
moreover
have C \text{ sorry}
\end{verbatim}
ultimately
have $A$ and $B$ and $C$.
next

Slightly more content (trivial bigstep reasoning):

have $A$ sorry
moreover
have $B$ sorry
moreover
have $C$ sorry
ultimately
have $A \land B \land C$ by blast
next

More ambitious bigstep reasoning involving structured results:

have $A \lor B \lor C$ sorry
moreover
{ assume $A$ have $R$ sorry }
moreover
{ assume $B$ have $R$ sorry }
moreover
{ assume $C$ have $R$ sorry }
ultimately
have $R$ by blast — “big-bang integration” of proof blocks (occasionally fragile)

end

1.3 Induction

1.3.1 Induction as Natural Deduction

In principle, induction is just a special case of Natural Deduction (see also §1.4). For example:

\begin{verbatim}
thm nat.induct
print_statement nat.induct

notepad
begin
  fix $n :: \text{nat}$
  have $P \ n$
  proof (rule nat.induct) — fragile rule application!
    show $P \ 0$ sorry
\end{verbatim}
next
  fix n :: nat
  assume P n
  show P (Suc n) sorry
qed
end

In practice, much more proof infrastructure is required. The proof method induct provides:

- implicit rule selection and robust instantiation
- context elements via symbolic case names
- support for rule-structured induction statements, with local parameters, premises, etc.

Example

The subsequent example combines the following proof patterns:

- outermost induction (over the datatype structure of natural numbers), to decompose the proof problem in top-down manner
- calculational reasoning (§1.2) to compose the result in each case
- solving local claims within the calculation by simplification

lemma
  fixes n :: nat
  shows (∑i=0..n. i) = n * (n + 1) div 2
CHAPTER 1. SYNOPSIS

proof (induct n)
case 0
  have (\sum i=0..0. i) = (0::nat) by simp
  also have \ldots = 0 * (0 + 1) div 2 by simp
  finally show ?case.
next
case (Suc n)
  have (\sum i=0..Suc n. i) = (\sum i=0..n. i) + (n + 1) by simp
  also have \ldots = n * (n + 1) div 2 + (n + 1) by (simp add: Suc.hyps)
  also have \ldots = (Suc n * (Suc n + 1)) div 2 by simp
  finally show ?case.
qed

This demonstrates how induction proofs can be done without having to consider the raw Natural Deduction structure.

1.3.2 Induction with local parameters and premises

Idea: Pure rule statements are passed through the induction rule. This achieves convenient proof patterns, thanks to some internal trickery in the induct method.

Important: Using compact HOL formulae with \forall \longrightarrow is a well-known anti-pattern! It would produce useless formal noise.

notepad
begin
  fix n :: nat
  fix P :: nat \Rightarrow bool
  fix Q :: 'a \Rightarrow nat \Rightarrow bool

  have P n
  proof (induct n)
    case 0
    show P 0 sorry
  next
    case (Suc n)
    from 'P n' show P (Suc n) sorry
  qed

  have A n \Rightarrow P n
  proof (induct n)
    case 0

from 'A 0' show P 0 sorry
next
case (Suc n)
from 'A n ⇒ P n'
and 'A (Suc n)' show P (Suc n) sorry
qed

have ∃x. Q x n
proof (induct n)
case 0
  show Q x 0 sorry
next
case (Suc n)
from '∀x. A x n' show Q x (Suc n) sorry

Local quantification admits arbitrary instances:

  note 'Q a n' and 'Q b n'
qed

end

1.3.3 Implicit induction context

The induct method can isolate local parameters and premises directly from the given statement. This is convenient in practical applications, but requires some understanding of what is going on internally (as explained above).

notepad
begin
  fix n :: nat
  fix Q :: 'a ⇒ nat ⇒ bool

  fix x :: 'a
  assume A x n
  then have Q x n
  proof (induct n arbitrary: x)
    case 0
    from 'A x 0' show Q x 0 sorry
  next
    case (Suc n)
    from '∀x. A x n ⇒ Q x n' — arbitrary instances can be produced here
    and 'A x (Suc n)' show Q x (Suc n) sorry
  qed
end
1.3.4 Advanced induction with term definitions

Induction over subexpressions of a certain shape are delicate to formalize. The Isar *induct* method provides infrastructure for this.

Idea: sub-expressions of the problem are turned into a defined induction variable; often accompanied with fixing of auxiliary parameters in the original expression.

```plaintext
notepad
begin
  fix a :: 'a ⇒ nat
  fix A :: nat ⇒ bool

assume A (a x)
then have P (a x)
proof (induct a x arbitrary: x)
case 0
  note prem = 'A (a x)'
  and defn = '0 = a x'
  show P (a x) sorry
next
  case (Suc n)
  note hyp = '∀x. n = a x ⇒ A (a x) ⇒ P (a x)'
  and prem = 'A (a x)'
  and defn = 'Suc n = a x'
  show P (a x) sorry
qed
end
```

1.4 Natural Deduction

1.4.1 Rule statements

Isabelle/Pure “theorems” are always natural deduction rules, which sometimes happen to consist of a conclusion only.

The framework connectives ∃ and ⇒⇒ indicate the rule structure declaratively. For example:

```plaintext
thm conjI
thm implI
thm nat.induct
```

The object-logic is embedded into the Pure framework via an implicit derivability judgment Trueprop :: bool ⇒ prop.
Thus any HOL formulae appears atomic to the Pure framework, while the
rule structure outlines the corresponding proof pattern.
This can be made explicit as follows:

```
notepad
begin
  write Trueprop (Tr)

  thm conjI
  thm impI
  thm nat.induct
end
```

Isar provides first-class notation for rule statements as follows.

```
print_statement conjI
print_statement impI
print_statement nat.induct
```

Examples

Introductions and eliminations of some standard connectives of the object-
logic can be written as rule statements as follows. (The proof “by blast”
serves as sanity check.)

```
lemma (P ⇒ False) ⇒ ¬ P by blast
lemma ¬ P ⇒ P ⇒ Q by blast

lemma P ⇒ Q ⇒ P ∧ Q by blast
lemma P ∧ Q ⇒ (P ⇒ Q ⇒ R) ⇒ R by blast

lemma P ⇒ P ∨ Q by blast
lemma Q ⇒ P ∨ Q by blast
lemma P ∨ Q ⇒ (P ⇒ R) ⇒ (Q ⇒ R) ⇒ R by blast

lemma (∀x. P x) ⇒ (∀x. P x) by blast
lemma (∀x. P x) ⇒ P x by blast

lemma P x ⇒ (∃x. P x) by blast
lemma (∃x. P x) ⇒ (∀x. P x ⇒ R) ⇒ R by blast

lemma x ∈ A ⇒ x ∈ B ⇒ x ∈ A ∩ B by blast
lemma x ∈ A ∩ B ⇒ (x ∈ A ⇒ x ∈ B ⇒ R) ⇒ R by blast

lemma x ∈ A ⇒ x ∈ A ∪ B by blast
```
lemma $x \in B \Rightarrow x \in A \cup B$ by blast
lemma $x \in A \cup B \Rightarrow (x \in A \Rightarrow R) \Rightarrow (x \in B \Rightarrow R) \Rightarrow R$ by blast

1.4.2 Isar context elements

We derive some results out of the blue, using Isar context elements and some explicit blocks. This illustrates their meaning wrt. Pure connectives, without goal states getting in the way.

begin
{fix $x$
 have $B \; x$ sorry}
have $\forall x. B \; x$ by fact

next
{
assume $A$
 have $B$ sorry
}
have $A \Rightarrow B$ by fact

next
{
def $x \equiv t$
 have $B \; x$ sorry
}
have $B \; t$ by fact

next
{
obtain $x :: 'a$ where $B \; x$ sorry
 have $C$ sorry
}
have $C$ by fact

end
CHAPTER 1. SYNOPSIS

1.4.3 Pure rule composition

The Pure framework provides means for:

- backward-chaining of rules by resolution
- closing of branches by assumption

Both principles involve higher-order unification of $\lambda$-terms modulo $\alpha \beta \eta$-equivalence (cf. Huet and Miller).

```
notepad
begin
  assume a: A and b: B
  thm conjI
  thm conjI [of A B] — instantiation
  thm conjI [of A B, OF a b] — instantiation and composition
  thm conjI [OF a b] — composition via unification (trivial)
  thm conjI [OF 'A' 'B']

  thm conjI [OF disjI1]
end
```

Note: Low-level rule composition is tedious and leads to unreadable / un-maintainable expressions in the text.

1.4.4 Structured backward reasoning

Idea: Canonical proof decomposition via fix / assume / show, where the body produces a natural deduction rule to refine some goal.

```
notepad
begin
  fix A B :: 'a => bool

  have \(\forall x. A x \implies B x\)
  proof
    fix x
    assume A x
    show B x sorry
  qed

  have \(\forall x. A x \implies B x\)
  proof
```


{  
  fix \(x\)  
  assume \(A \ x\)  
  show \(B \ x \) sorry  
} — implicit block structure made explicit

\textbf{note} ‘\(\forall x. A \ x \implies B \ x’\)

— side exit for the resulting rule

\texttt{qed}

\texttt{end}

\section{Structured rule application}

\textbf{Idea}: Previous facts and new claims are composed with a rule from the context (or background library).

\begin{verbatim}
notepad
begin
  assume \(r1: A \implies B \implies C\) — simple rule (Horn clause)

  have \(A\) sorry — prefix of facts via outer sub-proof
  then have \(C\)
  proof (\textit{rule} \(r1\))
    show \(B\) sorry — remaining rule premises via inner sub-proof
    \texttt{qed}

  have \(C\)
  proof (\textit{rule} \(r1\))
    show \(A\) sorry
    show \(B\) sorry
    \texttt{qed}

  have \(A\) and \(B\) sorry
  then have \(C\)
  proof (\textit{rule} \(r1\))
  \texttt{qed}

  have \(A\) and \(B\) sorry
  then have \(C\) \textit{by} (\textit{rule} \(r1\))

next

  assume \(r2: A \implies (\forall x. B1 \ x \implies B2 \ x) \implies C\) — nested rule

  have \(A\) sorry
\end{verbatim}
then have $C$
proof (rule r2)
  fix $x$
  assume $B1$ $x$
  show $B2$ $x$ sorry
qed

The compound rule premise $\forall x. B1 \ x \Rightarrow B2 \ x$ is better addressed via fix / assume / show in the nested proof body.

end

1.4.6 Example: predicate logic

Using the above principles, standard introduction and elimination proofs of predicate logic connectives of HOL work as follows.

notepad
begin
  have $A \rightarrow B$ and $A$ sorry
  then have $B$ ..

  have $A$ sorry
  then have $A \lor B$ ..

  have $B$ sorry
  then have $A \lor B$ ..

  have $A \lor B$ sorry
  then have $C$
proof
    assume $A$
    then show $C$ sorry
next
  assume $B$
  then show $C$ sorry
qed

  have $A$ and $B$ sorry
  then have $A \land B$ ..

  have $A \land B$ sorry
  then have $A$ ..

  have $A \land B$ sorry
then have \( B \) ..

have \( \text{False} \) sorry
then have \( A \) ..

have \( \text{True} \) ..

have \( \neg A \)
proof
  assume \( A \)
  then show \( \text{False} \) sorry
qed

have \( \neg A \) and \( A \) sorry
then have \( B \) ..

have \( \forall x. P \ x \)
proof
  fix \( x \)
  show \( P \ x \) sorry
qed

have \( \forall x. P \ x \) sorry
then have \( P \ a \) ..

have \( \exists x. P \ x \)
proof
  show \( P \ a \) sorry
qed

have \( \exists x. P \ x \) sorry
then have \( C \)
proof
  fix \( a \)
  assume \( P \ a \)
  show \( C \) sorry
qed

Less awkward version using \texttt{obtain}:

have \( \exists x. P \ x \) sorry
then obtain \( a \) where \( P \ a \) ..
end

Further variations to illustrate Isar sub-proofs involving \texttt{show}:
CHAPTER 1. SYNOPSIS

Example: set-theoretic operators

There is nothing special about logical connectives (\(\land, \lor, \forall, \exists\) etc.). Operators from set-theory or lattice-theory work analogously. It is only a matter of rule declarations in the library; rules can be also specified explicitly.

notepad
begin
  have \(A \land B\)
  proof — two strictly isolated subproofs
    show \(A\) sorry
  next
    show \(B\) sorry
  qed

  have \(A \land B\)
  proof — one simultaneous sub-proof
    show \(A\) and \(B\) sorry
  qed

  have \(A \land B\)
  proof — two subproofs in the same context
    show \(A\) sorry
    show \(B\) sorry
  qed

  have \(A \land B\)
  proof — swapped order
    show \(B\) sorry
    show \(A\) sorry
  qed

  have \(A \land B\)
  proof — sequential subproofs
    show \(A\) sorry
    show \(B\) using ‘\(A\)‘ sorry
  qed
end

Example: set-theoretic operators

There is nothing special about logical connectives (\(\land, \lor, \forall, \exists\) etc.). Operators from set-theory or lattice-theory work analogously. It is only a matter of rule declarations in the library; rules can be also specified explicitly.

notepad
begin
  have \(x \in A\) and \(x \in B\) sorry
  then have \(x \in A \cap B\) ..
have \( x \in A \) sorry
then have \( x \in A \cup B \)

have \( x \in B \) sorry
then have \( x \in A \cup B \)

have \( x \in A \cup B \) sorry
then have \( C \)
proof
  assume \( x \in A \)
  then show \( C \) sorry
next
  assume \( x \in B \)
  then show \( C \) sorry
qed

next
have \( x \in \bigcap A \)
proof
  fix \( a \)
  assume \( a \in A \)
  show \( x \in a \) sorry
qed

have \( x \in \bigcap A \) sorry
then have \( x \in a \)
proof
  show \( a \in A \) sorry
qed

have \( a \in A \) and \( x \in a \) sorry
then have \( x \in \bigcup A \)

have \( x \in \bigcup A \) sorry
then obtain \( a \) where \( a \in A \) and \( x \in a \)
end
CHAPTER 1. SYNOPSIS

1.5 Generalized elimination and cases

1.5.1 General elimination rules

The general format of elimination rules is illustrated by the following typical representatives:

- \texttt{thm \textit{ex}E} — local parameter
- \texttt{thm \textit{conj}E} — local premises
- \texttt{thm \textit{disj}E} — split into cases

Combining these characteristics leads to the following general scheme for elimination rules with cases:

- prefix of assumptions (or “major premises”)
- one or more cases that enable to establish the main conclusion in an augmented context

\begin{verbatim}
notepad begin
assume \textit{r}:
A1 \implies A2 \implies (* assumptions *)
(\forall x y. B1 x y \implies C1 x y \implies R) \implies (* case 1 *)
(\forall x y. B2 x y \implies C2 x y \implies R) \implies (* case 2 *)
R (* main conclusion *)

have A1 and A2 sorry
then have R proof (rule \textit{r})
fix x y
assume B1 x y and C1 x y
show \textit{thesis} sorry
next
fix x y
assume B2 x y and C2 x y
show \textit{thesis} sorry
qed
end
\end{verbatim}

Here \textit{thesis} is used to refer to the unchanged goal statement.
1.5.2 Rules with cases

Applying an elimination rule to some goal, leaves that unchanged but allows to augment the context in the sub-proof of each case. Isar provides some infrastructure to support this:

- native language elements to state eliminations
- symbolic case names
- method cases to recover this structure in a sub-proof

```
print_statement exE
print_statement conjE
print_statement disjE
```

```
lemma
  assumes A1 and A2 — assumptions
  obtains
    (case1) x y where B1 x y and C1 x y
    | (case2) x y where B2 x y and C2 x y
  sorry
```

Example

```
lemma tertium_non_datur:
  obtains
    (T) A
    | (F) ¬ A
  by blast

notepad
begin
  fix x y :: 'a
  have C
  proof (cases x = y rule: tertium_non_datur)
    case T
    from 'x = y' show ?thesis sorry
  next
    case F
    from 'x ≠ y' show ?thesis sorry
  qed
end
```
Example

Isabelle/HOL specification mechanisms (datatype, inductive, etc.) provide suitable derived cases rules.

```isar
datatype foo = Foo | Bar foo
```

```isar
notepad
begin
fix x :: foo
have C
proof (cases x)
  case Foo
  from 'x = Foo' show ?thesis sorry
next
  case (Bar a)
  from 'x = Bar a' show ?thesis sorry
qed
end
```

1.5.3 Obtaining local contexts

A single “case” branch may be inlined into Isar proof text via `obtain`. This proves \((\forall x. B x \Rightarrow thesis) \Rightarrow thesis\) on the spot, and augments the context afterwards.

```isar
notepad
begin
fix B :: 'a \Rightarrow bool

obtain x where B x sorry
note 'B x'

Conclusions from this context may not mention \(x\) again!

\{
  obtain x where B x sorry
  from 'B x' have C sorry
\}

note 'C'
end
```
Chapter 2

The Isabelle/Isar Framework

Isabelle/Isar [45, 46, 23, 49, 47] is intended as a generic framework for developing formal mathematical documents with full proof checking. Definitions and proofs are organized as theories. An assembly of theory sources may be presented as a printed document; see also chapter 4.

The main objective of Isar is the design of a human-readable structured proof language, which is called the “primary proof format” in Isar terminology. Such a primary proof language is somewhere in the middle between the extremes of primitive proof objects and actual natural language. In this respect, Isar is a bit more formalistic than Mizar [42, 39, 50], using logical symbols for certain reasoning schemes where Mizar would prefer English words; see [51] for further comparisons of these systems.

So Isar challenges the traditional way of recording informal proofs in mathematical prose, as well as the common tendency to see fully formal proofs directly as objects of some logical calculus (e.g. λ-terms in a version of type theory). In fact, Isar is better understood as an interpreter of a simple block-structured language for describing the data flow of local facts and goals, interspersed with occasional invocations of proof methods. Everything is reduced to logical inferences internally, but these steps are somewhat marginal compared to the overall bookkeeping of the interpretation process. Thanks to careful design of the syntax and semantics of Isar language elements, a formal record of Isar instructions may later appear as an intelligible text to the attentive reader.

The Isar proof language has emerged from careful analysis of some inherent virtues of the existing logical framework of Isabelle/Pure [33, 34], notably composition of higher-order natural deduction rules, which is a generalization of Gentzen’s original calculus [11]. The approach of generic inference systems in Pure is continued by Isar towards actual proof texts.

Concrete applications require another intermediate layer: an object-logic. Isabelle/HOL [26] (simply-typed set-theory) is being used most of the time; Isabelle/ZF [30] is less extensively developed, although it would probably fit better for classical mathematics.
In order to illustrate natural deduction in Isar, we shall refer to the background theory and library of Isabelle/HOL. This includes common notions of predicate logic, naive set-theory etc. using fairly standard mathematical notation. From the perspective of generic natural deduction there is nothing special about the logical connectives of HOL (\(\land\), \(\lor\), \(\forall\), \(\exists\), etc.), only the resulting reasoning principles are relevant to the user. There are similar rules available for set-theory operators (\(\cap\), \(\cup\), \(\bigcap\), \(\bigcup\), etc.), or any other theory developed in the library (lattice theory, topology etc.).

Subsequently we briefly review fragments of Isar proof texts corresponding directly to such general deduction schemes. The examples shall refer to set-theory, to minimize the danger of understanding connectives of predicate logic as something special.

The following deduction performs \(\cap\)-introduction, working forwards from assumptions towards the conclusion. We give both the Isar text, and depict the primitive rule involved, as determined by unification of the problem against rules that are declared in the library context.

\[
\text{assume } x \in A \text{ and } x \in B \\
\text{then have } x \in A \cap B \ldots
\]

Note that \textit{assume} augments the proof context, \textit{then} indicates that the current fact shall be used in the next step, and \textit{have} states an intermediate goal. The two dots “…” refer to a complete proof of this claim, using the indicated facts and a canonical rule from the context. We could have been more explicit here by spelling out the final proof step via the \textit{by} command:

\[
\text{assume } x \in A \text{ and } x \in B \\
\text{then have } x \in A \cap B \text{ by (rule IntI)}
\]

The format of the \(\cap\)-introduction rule represents the most basic inference, which proceeds from given premises to a conclusion, without any nested proof context involved.

The next example performs backwards introduction on \(\bigcap A\), the intersection of all sets within a given set. This requires a nested proof of set membership within a local context, where \(A\) is an arbitrary-but-fixed member of the collection:

\[
\text{have } x \in \bigcap A \\
\text{proof} \\
\text{fix } A \\
\text{assume } A \in A \\
\text{show } x \in A \langle \text{proof} \rangle \\
\text{qed}
\]

\[
[A][A \in A] \\
\vdots \\
\frac{x \in A}{x \in \bigcap A}
\]
This Isar reasoning pattern again refers to the primitive rule depicted above. The system determines it in the “proof” step, which could have been spelt out more explicitly as “proof (rule InterI)”. Note that the rule involves both a local parameter \( A \) and an assumption \( A \in \mathcal{A} \) in the nested reasoning. This kind of compound rule typically demands a genuine sub-proof in Isar, working backwards rather than forwards as seen before. In the proof body we encounter the fix-assume-show outline of nested sub-proofs that is typical for Isar. The final show is like have followed by an additional refinement of the enclosing claim, using the rule derived from the proof body.

The next example involves \( \bigcup \mathcal{A} \), which can be characterized as the set of all \( x \) such that \( \exists A. \ x \in A \land A \in \mathcal{A} \). The elimination rule for \( x \in \bigcup \mathcal{A} \) does not mention \( \exists \) and \( \land \) at all, but admits to obtain directly a local \( A \) such that \( x \in A \) and \( A \in \mathcal{A} \) hold. This corresponds to the following Isar proof and inference rule, respectively:

\[
\begin{align*}
\text{assume } x & \in \bigcup \mathcal{A} \\
\text{then have } & C \\
\text{proof} \\
& \text{fix } A \\
& \text{assume } x \in A \text{ and } A \in \mathcal{A} \\
& \text{show } C \langle \text{proof} \rangle \\
& \text{qed}
\end{align*}
\]

\[\begin{array}{c}
[A][x \in A, \ A \in \mathcal{A}] \\
\vdots \\
x \in \bigcup \mathcal{A} \\
C \\
\hline
\end{array}\]

Although the Isar proof follows the natural deduction rule closely, the text reads not as natural as anticipated. There is a double occurrence of an arbitrary conclusion \( C \), which represents the final result, but is irrelevant for now. This issue arises for any elimination rule involving local parameters. Isar provides the derived language element obtain, which is able to perform the same elimination proof more conveniently:

\[
\begin{align*}
\text{assume } x & \in \bigcup \mathcal{A} \\
\text{then obtain } & A \text{ where } x \in A \text{ and } A \in \mathcal{A} .. \\
\text{qed}
\end{align*}
\]

Here we avoid to mention the final conclusion \( C \) and return to plain forward reasoning. The rule involved in the “..” proof is the same as before.

### 2.1 The Pure framework

The Pure logic [33, 34] is an intuitionistic fragment of higher-order logic [9]. In type-theoretic parlance, there are three levels of \( \lambda \)-calculus with corresponding arrows \( \Rightarrow/\\land/\implies \):
α ⇒ β syntactic function space (terms depending on terms)
∧x. B(x) universal quantification (proofs depending on terms)
A ⇒ B implication (proofs depending on proofs)

Here only the types of syntactic terms, and the propositions of proof terms have been shown. The λ-structure of proofs can be recorded as an optional feature of the Pure inference kernel [5], but the formal system can never depend on them due to proof irrelevance.

On top of this most primitive layer of proofs, Pure implements a generic calculus for nested natural deduction rules, similar to [40]. Here object-logic inferences are internalized as formulae over ∧ and ⇒. Combining such rule statements may involve higher-order unification [32].

2.1.1 Primitive inferences

Term syntax provides explicit notation for abstraction \( \lambda x :: \alpha. \ b(x) \) and application \( b \ a \), while types are usually implicit thanks to type-inference; terms of type prop are called propositions. Logical statements are composed via \( \forall x :: \alpha. \ B(x) \) and \( A \Rightarrow B \). Primitive reasoning operates on judgments of the form \( \Gamma \vdash \varphi \), with standard introduction and elimination rules for ∧ and ⇒ that refer to fixed parameters \( x_1, \ldots, x_m \) and hypotheses \( A_1, \ldots, A_n \) from the context \( \Gamma \); the corresponding proof terms are left implicit. The subsequent inference rules define \( \Gamma \vdash \varphi \) inductively, relative to a collection of axioms:

\[
\begin{align*}
(A \text{ axiom}) & \quad \Gamma \vdash A \\
\hline
A \vdash A
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash B(x) & \quad x \notin \Gamma \\
\hline
\Gamma \vdash \forall x. \ B(x)
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash B & \quad \Gamma_1 \vdash A \Rightarrow B \quad \Gamma_2 \vdash A \\
\hline
\Gamma_1 \cup \Gamma_2 \vdash B
\end{align*}
\]

Furthermore, Pure provides a built-in equality \( \equiv :: \alpha \Rightarrow \alpha \Rightarrow \text{prop} \) with axioms for reflexivity, substitution, extensionality, and \( \alpha\beta\eta \)-conversion on \( \lambda \)-terms.

An object-logic introduces another layer on top of Pure, e.g. with types \( i \) for individuals and \( o \) for propositions, term constants \( \text{Trueprop} :: o \Rightarrow \text{prop} \) as (implicit) derivability judgment and connectives like \( \& :: o \Rightarrow o \Rightarrow o \) or \( \forall :: (i \Rightarrow o) \Rightarrow o \), and axioms for object-level rules such as \( \text{conjI} : A \Rightarrow \)
Chapter 2. The Isabelle/Isar Framework

\[ B \implies A \land B \text{ or } allI: (\land x. B x) \implies \forall x. B x. \] Derived object rules are represented as theorems of Pure. After the initial object-logic setup, further axiomatizations are usually avoided; plain definitions and derived principles are used exclusively.

2.1.2 Reasoning with rules

Primitive inferences mostly serve foundational purposes. The main reasoning mechanisms of Pure operate on nested natural deduction rules expressed as formulae, using \( \land \) to bind local parameters and \( \implies \) to express entailment. Multiple parameters and premises are represented by repeating these connectives in a right-associative manner.

Since \( \land \) and \( \implies \) commute thanks to the theorem \((A \implies (\land x. B x)) \equiv (\land x. A \implies B x)\), we may assume w.l.o.g. that rule statements always observe the normal form where quantifiers are pulled in front of implications at each level of nesting. This means that any Pure proposition may be presented as a Hereditary Harrop Formula \[19\] which is of the form \( \land x_1 \ldots x_m. H_1 \implies \ldots H_n \implies A \) for \( m, n \geq 0 \), and \( A \) atomic, and \( H_1, \ldots, H_n \) being recursively of the same format. Following the convention that outermost quantifiers are implicit, Horn clauses \( A_1 \implies \ldots A_n \implies A \) are a special case of this.

For example, \( \cap \)-introduction rule encountered before is represented as a Pure theorem as follows:

\[ \text{IntI: } x \in A \implies x \in B \implies x \in A \cap B \]

This is a plain Horn clause, since no further nesting on the left is involved. The general \( \cap \)-introduction corresponds to a Hereditary Harrop Formula with one additional level of nesting:

\[ \text{InterI: } (\land A. A \in A \implies x \in A) \implies x \in \bigcap A \]

Goals are also represented as rules: \( A_1 \implies \ldots A_n \implies C \) states that the sub-goals \( A_1, \ldots, A_n \) entail the result \( C \); for \( n = 0 \) the goal is finished. To allow \( C \) being a rule statement itself, we introduce the protective marker \# :: prop \( \Rightarrow \) prop, which is defined as identity and hidden from the user. We initialize and finish goal states as follows:

\[ \frac{C \implies \#C \text{ (init)}}{C \implies \#C \text{ (finish)}} \]
Goal states are refined in intermediate proof steps until a finished form is
achieved. Here the two main reasoning principles are \textit{resolution}, for back-
chaining a rule against a sub-goal (replacing it by zero or more sub-goals),
and \textit{assumption}, for solving a sub-goal (finding a short-circuit with local
assumptions). Below \( \pi \) stands for \( x_1, \ldots, x_n \) (\( n \geq 0 \)).

\[
\text{rule: } \overline{A} \pi \rightarrow B \pi \\
\text{goal: } (\land \pi. H \pi \rightarrow B' \pi) \rightarrow C \\
\text{goal unifier: } (\lambda \pi. B (\overline{a} \pi)) \theta = B' \theta \qquad \text{(resolution)}
\]

\[
\text{goal: } (\land \pi. H \pi \rightarrow A \pi) \rightarrow C \\
\text{assm unifier: } A \theta = H_i \theta \quad \text{(for some } H_i) \qquad \text{(assumption)}
\]

The following trace illustrates goal-oriented reasoning in Isabelle/Pure:

\[
\begin{align*}
(A \land B \rightarrow B \land A) \rightarrow \#(A \land B \rightarrow B \land A) \quad \text{(init)} \\
(A \land B \rightarrow B) \rightarrow (A \land B \rightarrow A) \rightarrow \# \ldots \quad \text{(resolution } B \rightarrow A \rightarrow B \land A) \\
(A \land B \rightarrow A \land B) \rightarrow (A \land B \rightarrow A) \rightarrow \# \ldots \quad \text{(resolution } A \land B \rightarrow B) \\
(A \land B \rightarrow A) \rightarrow \# \ldots \quad \text{(assumption)} \\
(A \land B \rightarrow B \land A) \rightarrow \# \ldots \quad \text{(resolution } A \land B \rightarrow A) \\
\# \ldots \quad \text{(assumption)} \\
A \land B \rightarrow B \land A \quad \text{(finish)}
\end{align*}
\]

Compositions of \textit{assumption} after \textit{resolution} occurs quite often, typically in
elimination steps. Traditional Isabelle tactics accommodate this by a com-
bined \textit{elim-resolution} principle. In contrast, Isar uses a slightly more refined
combination, where the assumptions to be closed are marked explicitly, using
again the protective marker \#:

\[
\begin{align*}
\text{sub-proof: } \overline{G} \pi \rightarrow B \pi \\
\text{goal: } (\land \pi. H \pi \rightarrow B' \pi) \rightarrow C \\
\text{goal unifier: } (\lambda \pi. B (\overline{a} \pi)) \theta = B' \theta \\
\text{assm unifiers: } (\lambda \pi. G_j (\overline{a} \pi)) \theta = \#H_i \theta \\
\quad \text{(for each marked } G_j \text{ some } \#H_i) \\
\text{refinement: } (\land \pi. H \pi \rightarrow \overline{G'} (\overline{a} \pi)) \theta \rightarrow C \theta
\end{align*}
\]

Here the \textit{sub-proof} rule stems from the main \textbf{fix-assume-show} outline of
Isar (cf. §2.2.3): each assumption indicated in the text results in a marked
premise $G$ above. The marking enforces resolution against one of the sub-
goal’s premises. Consequently, fix-assume-show enables to fit the result of
a sub-proof quite robustly into a pending sub-goal, while maintaining a good
measure of flexibility.

2.2 The Isar proof language

Structured proofs are presented as high-level expressions for composing enti-
ties of Pure (propositions, facts, and goals). The Isar proof language allows
to organize reasoning within the underlying rule calculus of Pure, but Isar is
not another logical calculus!

Isar is an exercise in sound minimalism. Approximately half of the language
is introduced as primitive, the rest defined as derived concepts. The following
grammar describes the core language (category proof), which is embedded
into theory specification elements such as theorem; see also §2.2.2 for the
separate category statement.

\[
\begin{align*}
\text{theory-stmt} & = \text{theorem statement proof} \mid \text{definition} \ldots \mid \ldots \\
\text{proof} & = \text{prfx}^* \text{ proof method}^2 \text{ stmt}^* \text{ qed method}^2 \\
\text{prfx} & = \text{using facts} \\
& \mid \text{unfolding facts} \\
\text{stmt} & = \{ \text{ stmt}^* \} \\
& \mid \text{next} \\
& \mid \text{note name = facts} \\
& \mid \text{let term = term} \\
& \mid \text{fix var}^+ \\
& \mid \text{assume } \langle \text{inference} \rangle \text{ name: props} \\
& \mid \text{then }^? \text{ goal} \\
\text{goal} & = \text{have name: props proof} \\
& \mid \text{show name: props proof}
\end{align*}
\]

Simultaneous propositions or facts may be separated by the and keyword.

The syntax for terms and propositions is inherited from Pure (and the object-
logic). A pattern is a term with schematic variables, to be bound by higher-
order matching.

Facts may be referenced by name or proposition. For example, the result of

\text{“have } a: A \langle \text{proof} \rangle \text{” becomes available both as } a \text{ and } ‘A’. Moreover, fact
expressions may involve attributes that modify either the theorem or the
background context. For example, the expression “a [OF b]” refers to the
composition of two facts according to the resolution inference of §2.1.2, while “a [intro]” declares a fact as introduction rule in the context.

The special fact called “this” always refers to the last result, as produced by note, assume, have, or show. Since note occurs frequently together with then we provide some abbreviations:

\[
\begin{align*}
\text{from } a & \equiv \text{ note } a \text{ then} \\
\text{with } a & \equiv \text{ from } a \text{ and this}
\end{align*}
\]

The method category is essentially a parameter and may be populated later. Methods use the facts indicated by then or using, and then operate on the goal state. Some basic methods are predefined: “−” leaves the goal unchanged, “this” applies the facts as rules to the goal, “rule” applies the facts to another rule and the result to the goal (both “this” and “rule” refer to resolution of §2.1.2). The secondary arguments to “rule” may be specified explicitly as in “(rule a)”, or picked from the context. In the latter case, the system first tries rules declared as elim or dest, followed by those declared as intro.

The default method for proof is “rule” (arguments picked from the context), for qed it is “−”. Further abbreviations for terminal proof steps are “by method\textsubscript{1} method\textsubscript{2}” for “proof method\textsubscript{1} qed method\textsubscript{2}”, and “..” for “by rule, and “.” for “by this”. The unfolding element operates directly on the current facts and goal by applying equalities.

Block structure can be indicated explicitly by “{ ... }”, although the body of a sub-proof already involves implicit nesting. In any case, next jumps into the next section of a block, i.e. it acts like closing an implicit block scope and opening another one; there is no direct correspondence to subgoals here.

The remaining elements fix and assume build up a local context (see §2.2.1), while show refines a pending sub-goal by the rule resulting from a nested sub-proof (see §2.2.3). Further derived concepts will support calculational reasoning (see §2.2.4).

### 2.2.1 Context elements

In judgments $\Gamma \vdash \varphi$ of the primitive framework, $\Gamma$ essentially acts like a proof context. Isar elaborates this idea towards a higher-level notion, with additional information for type-inference, term abbreviations, local facts, hypotheses etc.

The element fix $x :: \alpha$ declares a local parameter, i.e. an arbitrary-but-fixed entity of a given type; in results exported from the context, $x$ may become
anything. The \texttt{assume \textless inference\textgreater} element provides a general interface to hypotheses: “\texttt{assume \textless inference\textgreater } A” produces $A \vdash A$ locally, while the included inference tells how to discharge $A$ from results $A \vdash B$ later on. There is no user-syntax for \texttt{\textless inference\textgreater}, i.e. it may only occur internally when derived commands are defined in ML.

At the user-level, the default inference for \texttt{assume} is \texttt{discharge} as given below. The additional variants \texttt{presume} and \texttt{def} are defined as follows:

\begin{align*}
\texttt{presume } A &\equiv \texttt{assume } \textless \text{weak-discharge}\textgreater A \\
\texttt{def } x \equiv a &\equiv \texttt{fix } x \texttt{ assume } \textless \text{expansion}\textgreater x \equiv a
\end{align*}

\begin{align*}
\Gamma \vdash B \\
\Gamma - A \vdash \#A \implies B &\quad \text{(discharge)} \\
\Gamma \vdash B \\
\Gamma - A \vdash A \implies B &\quad \text{(weak-discharge)} \\
\Gamma \vdash B x \\
\Gamma - (x \equiv a) \vdash B a &\quad \text{(expansion)}
\end{align*}

Note that \texttt{discharge} and \texttt{weak-discharge} differ in the marker for $A$, which is relevant when the result of a \texttt{fix-assume-show} outline is composed with a pending goal, cf. §2.2.3.

The most interesting derived context element in Isar is \texttt{obtain} [46, §5.3], which supports generalized elimination steps in a purely forward manner. The \texttt{obtain} command takes a specification of parameters $\pi$ and assumptions $\overline{A}$ to be added to the context, together with a proof of a case rule stating that this extension is conservative (i.e. may be removed from closed results later on):

\begin{verbatim}
⟨facts⟩ obtain π where \overline{A} π ⟨proof⟩ ≡ 
  have case: \forall thesis. (\exists π. \overline{A} π ⇒ thesis) ⇒ thesis
  proof –
  fix thesis
  assume [intro]: \forall π. \overline{A} π ⇒ thesis
  show thesis using ⟨facts⟩ ⟨proof⟩
  qed
  fix π assume \textless elimination case\textgreater \overline{A} π

  case: \Gamma \vdash \forall thesis. (\exists π. \overline{A} π ⇒ thesis) ⇒ thesis
  result: \Gamma \cup \overline{A} π \vdash B \\
  \Gamma \vdash B
\end{verbatim}
Here the name “thesis” is a specific convention for an arbitrary-but-fixed proposition; in the primitive natural deduction rules shown before we have occasionally used $C$. The whole statement of “obtain $x$ where $A x$” may be read as a claim that $A x$ may be assumed for some arbitrary-but-fixed $x$. Also note that “obtain $A$ and $B$” without parameters is similar to “have $A$ and $B$”, but the latter involves multiple sub-goals.

The subsequent Isar proof texts explain all context elements introduced above using the formal proof language itself. After finishing a local proof within a block, we indicate the exported result via note.

\begin{verbatim}
{ 
  fix x 
  have B x ⟨proof⟩ 
} 

note '⋀x. B x' 

{ 
  def x ≡ a 
  have B x ⟨proof⟩ 
} 

note 'B a'
\end{verbatim}

This illustrates the meaning of Isar context elements without goals getting in between.

### 2.2.2 Structured statements

The category statement of top-level theorem specifications is defined as follows:

\[
\begin{align*}
\text{statement} & \equiv \text{name: props and} \ldots \\
& \quad \mid \text{context* conclusion} \\
\text{context} & \equiv \text{fixes vars and} \ldots \\
& \quad \mid \text{assumes name: props and} \ldots \\
\text{conclusion} & \equiv \text{shows name: props and} \ldots \\
& \quad \mid \text{obtains vars and} \ldots \text{where name: props and} \ldots \\
& \quad \mid \ldots
\end{align*}
\]

A simple statement consists of named propositions. The full form admits local context elements followed by the actual conclusions, such as “fixes $x$ assumes $A x$ shows $B x$”. The final result emerges as a Pure rule after discharging the context: \(\forall x. A x \implies B x\).
CHAPTER 2. THE ISABELLE/ISAR FRAMEWORK

The obtains variant is another abbreviation defined below; unlike obtain (cf. §2.2.1) there may be several “cases” separated by “|”, each consisting of several parameters (vars) and several premises (props). This specifies multi-branch elimination rules.

\texttt{obtains } x \texttt{ where } A x \quad | \quad \ldots \quad \equiv \quad \texttt{fixes } \texttt{thesis} \quad \texttt{assumes } \texttt{[intro]: } \wedge \tau. A \tau \Longrightarrow \texttt{thesis} \quad \texttt{and} \quad \ldots \quad \texttt{shows } \texttt{thesis}

Presenting structured statements in such an “open” format usually simplifies the subsequent proof, because the outer structure of the problem is already laid out directly. E.g., consider the following canonical patterns for \texttt{shows} and \texttt{obtains}, respectively:

\texttt{theorem} \\
\texttt{fixes } x \texttt{ and } y \\
\texttt{assumes } A x \texttt{ and } B y \\
\texttt{shows } C x y \\
\texttt{proof} – \\
\texttt{from } 'A x' \texttt{ and } 'B y' \\
\texttt{show } C x y \langle \texttt{proof} \rangle \\
\texttt{qed}

\texttt{theoremc} \\
\texttt{obtains } x \texttt{ and } y \\
\texttt{where } A x \texttt{ and } B y \\
\texttt{proof} – \\
\texttt{have } A a \texttt{ and } B b \langle \texttt{proof} \rangle \\
\texttt{then show } \texttt{thesis} \ldots \\
\texttt{qed}

Here local facts ‘A x’ and ‘B y’ are referenced immediately; there is no need to decompose the logical rule structure again. In the second proof the final “\texttt{then show } \texttt{thesis} \ldots” involves the local rule case $\wedge x. A x \Longrightarrow B y \Longrightarrow \texttt{thesis}$ for the particular instance of terms $a$ and $b$ produced in the body.

2.2.3 Structured proof refinement

By breaking up the grammar for the Isar proof language, we may understand a proof text as a linear sequence of individual proof commands. These are interpreted as transitions of the Isar virtual machine (Isar/VM), which operates on a block-structured configuration in single steps. This allows users to write proof texts in an incremental manner, and inspect intermediate configurations for debugging.

The basic idea is analogous to evaluating algebraic expressions on a stack machine: $(a + b) \cdot c$ then corresponds to a sequence of single transitions for each symbol $(, a, +, b, ), \cdot, c$. In Isar the algebraic values are facts or goals, and the operations are inferences.

The Isar/VM state maintains a stack of nodes, each node contains the local proof context, the linguistic mode, and a pending goal (optional). The mode
determines the type of transition that may be performed next, it essentially alternates between forward and backward reasoning, with an intermediate stage for chained facts (see figure 2.1).

For example, in state mode Isar acts like a mathematical scratch-pad, accepting declarations like fix, assume, and claims like have, show. A goal statement changes the mode to prove, which means that we may now refine the problem via unfolding or proof. Then we are again in state mode of a proof body, which may issue show statements to solve pending sub-goals. A concluding qed will return to the original state mode one level upwards. The subsequent Isar/VM trace indicates block structure, linguistic mode, goal state, and inferences:

\[
\begin{align*}
\text{have } A \rightarrow B & \quad \text{begin prove} \quad (A \rightarrow B) \quad \#(A \rightarrow B) \quad \text{(init)} \\
\text{proof} & \quad \text{state} \quad (A \Rightarrow B) \quad \#(A \rightarrow B) \quad \text{(resolution impI)} \\
\text{assume } A & \quad \text{state} \\
\text{show } B & \quad \text{begin prove} \\
\langle \text{proof} \rangle & \quad \text{end state} \quad \#(A \rightarrow B) \quad \text{(refinement } #A \Rightarrow B) \\
\text{qed} & \quad \text{end state} \quad A \rightarrow B \quad \text{(finish)}
\end{align*}
\]

Here the refinement inference from §2.1.2 mediates composition of Isar sub-proofs nicely. Observe that this principle incorporates some degree of freedom in proof composition. In particular, the proof body allows parameters and assumptions to be re-ordered, or commuted according to Hereditary Harrop Form. Moreover, context elements that are not used in a sub-proof may be omitted altogether. For example:
Such “peephole optimizations” of Isar texts are practically important to improve readability, by rearranging contexts elements according to the natural flow of reasoning in the body, while still observing the overall scoping rules.

This illustrates the basic idea of structured proof processing in Isar. The main mechanisms are based on natural deduction rule composition within the Pure framework. In particular, there are no direct operations on goal states within the proof body. Moreover, there is no hidden automated reasoning involved, just plain unification.

### 2.2.4 Calculational reasoning

The existing Isar infrastructure is sufficiently flexible to support calculational reasoning (chains of transitivity steps) as derived concept. The generic proof elements introduced below depend on rules declared as `trans` in the context. It is left to the object-logic to provide a suitable rule collection for mixed relations of `=`, `<`, `≤`, `⊂`, `⊆` etc. Due to the flexibility of rule composition (§2.1.2), substitution of equals by equals is covered as well, even substitution of inequalities involving monotonicity conditions; see also [46, §6] and [4].

The generic calculational mechanism is based on the observation that rules such as `trans`: \( x = y \implies y = z \implies x = z \) proceed from the premises towards the conclusion in a deterministic fashion. Thus we may reason in forward mode, feeding intermediate results into rules selected from the context. The course of reasoning is organized by maintaining a secondary fact called “calculation”, apart from the primary “this” already provided by the Isar primitives. In the definitions below, `OF` refers to `resolution` (§2.1.2) with multiple rule arguments, and `trans` represents to a suitable rule from
The start of a calculation is determined implicitly in the text: here \texttt{also} sets \texttt{calculation} to the current result; any subsequent occurrence will update \texttt{calculation} by combination with the next result and a transitivity rule. The calculational sequence is concluded via \texttt{finally}, where the final result is exposed for use in a concluding claim.

Here is a canonical proof pattern, using \texttt{have} to establish the intermediate results:

\begin{verbatim}
  have \texttt{a = b sorry}
  also have \ldots = \texttt{c sorry}
  also have \ldots = \texttt{d sorry}
  finally have \texttt{a = d}.
\end{verbatim}

The term “…” above is a special abbreviation provided by the Isabelle/Isar syntax layer: it statically refers to the right-hand side argument of the previous statement given in the text. Thus it happens to coincide with relevant sub-expressions in the calculational chain, but the exact correspondence is dependent on the transitivity rules being involved.

Symmetry rules such as \( x = y \implies y = x \) are like transitivities with only one premise. Isar maintains a separate rule collection declared via the \texttt{sym} attribute, to be used in fact expressions “\( a \ [\text{symmetric}] \)”, or single-step proofs “\texttt{assume x = y then have y = x ..}.”

\section{Example: First-Order Logic}

\begin{verbatim}
theory First_Order_Logic
imports Base
begin

In order to commence a new object-logic within Isabelle/Pure we introduce abstract syntactic categories \( i \) for individuals and \( o \) for object-propositions. The latter is embedded into the language of Pure propositions by means of a separate judgment.

\texttt{typedecl i}
\texttt{typedecl o}
\end{verbatim}
CHAPTER 2. THE ISABELLE/ISAR FRAMEWORK

judgment

\textit{Trueprop} :: o \Rightarrow prop \quad (\_\_ 5)

Note that the object-logic judgement is implicit in the syntax: writing \(A\) produces \textit{Trueprop} \(A\) internally. From the Pure perspective this means “\(A\) is derivable in the object-logic”.

2.3.1 Equational reasoning

Equality is axiomatized as a binary predicate on individuals, with reflexivity as introduction, and substitution as elimination principle. Note that the latter is particularly convenient in a framework like Isabelle, because syntactic congruences are implicitly produced by unification of \(B x\) against expressions containing occurrences of \(x\).

axiomatization

\textit{equal} :: i \Rightarrow i \Rightarrow o \quad (\text{infix} = 50)

where

\textit{refl} [intro]: \(x = x\) and
\textit{subst} [elim]: \(x = y \Rightarrow B x \Rightarrow B y\)

Substitution is very powerful, but also hard to control in full generality. We derive some common symmetry / transitivity schemes of \textit{equal} as particular consequences.

\textbf{theorem \textit{sym} [sym]:}
\begin{itemize}
  \item \textbf{assumes} \(x = y\)
  \item \textbf{shows} \(y = x\)
\end{itemize}

\textbf{proof} –
\begin{itemize}
  \item \textbf{have} \(x = x\) ..
  \item \textbf{with} ‘\(x = y\)’ \textbf{show} \(y = x\) ..
\end{itemize}
\textbf{qed}

\textbf{theorem \textit{forw} subst} [trans]:
\begin{itemize}
  \item \textbf{assumes} \(y = x\) and \(B x\)
  \item \textbf{shows} \(B y\)
\end{itemize}

\textbf{proof} –
\begin{itemize}
  \item \textbf{from} ‘\(y = x\)’ \textbf{have} \(x = y\) ..
  \item \textbf{from} \textbf{this} and ‘\(B x\)’ \textbf{show} \(B y\) ..
\end{itemize}
\textbf{qed}

\textbf{theorem \textit{back} subst} [trans]:
\begin{itemize}
  \item \textbf{assumes} \(B x\) and \(x = y\)
\end{itemize}
shows \( B y \)
proof –
  from \('x = y' and 'B x'\)
  show \( B y \).
qed

theorem \( \text{trans} [\text{trans}] \):
assumes \( x = y \) and \( y = z \)
shows \( x = z \)
proof –
  from \('y = z' and 'x = y'\)
  show \( x = z \).
qed

\subsection{2.3.2 Basic group theory}

As an example for equational reasoning we consider some bits of group theory. The subsequent locale definition postulates group operations and axioms; we also derive some consequences of this specification.

locale \( \text{group} = \)
  fixes \( \text{prod} :: i \Rightarrow i \Rightarrow i \) (\( \text{infix} \circ 70 \))
  and \( \text{inv} :: i \Rightarrow i \) (\( (\_^{-1}) [1000] 999 \))
  and \( \text{unit} :: i \) (1)
assumes \( \text{assoc} : (x \circ y) \circ z = x \circ (y \circ z) \)
  and \( \text{left_unit} : 1 \circ x = x \)
  and \( \text{left_inv} : x^{-1} \circ x = 1 \)
begin

theorem \( \text{right_inv} : x \circ x^{-1} = 1 \)
proof –
  have \( x \circ x^{-1} = 1 \circ (x \circ x^{-1}) \) by (rule \text{left_unit} [\text{symmetric}])
  also have \( (1 \circ x) \circ x^{-1} \) by (rule \text{assoc} [\text{symmetric}])
  also have \( 1 = (x^{-1})^{-1} \circ x^{-1} \) by (rule \text{left_inv} [\text{symmetric}])
  also have \( (x^{-1})^{-1} \circ x = (x^{-1})^{-1} \circ (x^{-1} \circ x) \) by (rule \text{assoc})
  also have \( x^{-1} \circ x = 1 \) by (rule \text{left_inv})
  also have \( ((x^{-1})^{-1} \circ \ldots) \circ x^{-1} = (x^{-1})^{-1} \circ (1 \circ x^{-1}) \) by (rule \text{assoc})
  also have \( 1 \circ x^{-1} = x^{-1} \) by (rule \text{left_unit})
  also have \( (x^{-1})^{-1} \circ \ldots = 1 \) by (rule \text{left_inv})
  finally show \( x \circ x^{-1} = 1 \).
qed

theorem \( \text{right_unit} : x \circ 1 = x \)
proof –

have $1 = x^{-1} \circ x$ by (rule left_inv [symmetric])
also have $x \circ \ldots = (x \circ x^{-1}) \circ x$ by (rule assoc [symmetric])
also have $x \circ x^{-1} = 1$ by (rule right_inv)
also have $\ldots \circ x = x$ by (rule left_unit)
finally show $x \circ 1 = x$.

qed

Reasoning from basic axioms is often tedious. Our proofs work by producing various instances of the given rules (potentially the symmetric form) using the pattern “have \textit{eq} by (rule r)” and composing the chain of results via also/finally. These steps may involve any of the transitivity rules declared in §2.3.1, namely trans in combining the first two results in right_inv and in the final steps of both proofs, forw subst in the first combination of right_unit, and back subst in all other calculational steps.

Occasional substitutions in calculations are adequate, but should not be over-emphasized. The other extreme is to compose a chain by plain transitivity only, with replacements occurring always in topmost position. For example:

have $x \circ 1 = x \circ (x^{-1} \circ x)$ unfolding left_inv ..
also have $\ldots = (x \circ x^{-1}) \circ x$ unfolding assoc ..
also have $\ldots = 1 \circ x$ unfolding right_inv ..
also have $\ldots = x$ unfolding left_unit ..
finally have $x \circ 1 = x$.

Here we have re-used the built-in mechanism for unfolding definitions in order to normalize each equational problem. A more realistic object-logic would include proper setup for the Simplifier (§9.3), the main automated tool for equational reasoning in Isabelle. Then “unfolding left_inv ..” would become “by (simp only: left_inv)” etc.

end

2.3.3 Propositional logic

We axiomatize basic connectives of propositional logic: implication, disjunction, and conjunction. The associated rules are modeled after Gentzen’s system of Natural Deduction [11].

axiomatization

imp :: $o \Rightarrow o \Rightarrow o$ (infixr $\rightarrow$ 25) where

impI [intro]: $(A \Rightarrow B) \Rightarrow A \Rightarrow B$ and
impD [dest]: $(A \Rightarrow B) \Rightarrow A \Rightarrow B$

axiomatization
**disj ::** \( o \Rightarrow o \Rightarrow o \)** (infixr \( \lor \) 30)** where
\[
disj_1 \ [\text{intro}]: \ A \Rightarrow A \lor B \ \text{and}
disj_2 \ [\text{intro}]: \ B \Rightarrow A \lor B \ \text{and}
disjE \ [\text{elim}]: \ A \lor B \Rightarrow (A \Rightarrow C) \Rightarrow (B \Rightarrow C) \Rightarrow C
\]

**axiomatization**
\[
\text{conj} :: \ o \Rightarrow o \Rightarrow o \ \text{**(infixr \( \land \) 35)** where
\[
\text{conj} \ [\text{intro}]: \ A \Rightarrow B \Rightarrow A \land B \ \text{and}
\text{conjD}_1: \ A \land B \Rightarrow A \ \text{and}
\text{conjD}_2: \ A \land B \Rightarrow B
\]

The conjunctive destructions have the disadvantage that decomposing \( A \land B \) involves an immediate decision which component should be projected. The more convenient simultaneous elimination \( A \land B \Rightarrow (A \Rightarrow B \Rightarrow C) \Rightarrow C \) can be derived as follows:

**theorem** **conjE [elim]:**
\[
\text{assumes} \ A \land B
\text{obtains} \ A \text{ and } B
\]
\[
\text{proof}
\text{from } 'A \land B' \text{ show } A \text{ by } (\text{rule conjD}_1)
\text{from } 'A \land B' \text{ show } B \text{ by } (\text{rule conjD}_2)
\]
\[
\text{qed}
\]

Here is an example of swapping conjuncts with a single intermediate elimination step:
\[
\text{assume } A \land B
\text{then obtain } B \text{ and } A \ \ldots
\text{then have } B \land A \ \ldots
\]

Note that the analogous elimination rule for disjunction \“\text{assumes } A \lor B \\text{obtains } A \mid B\”\ coincides with the original axiomatization of \( \text{disjE} \).

We continue propositional logic by introducing absurdity with its characteristic elimination. Plain truth may then be defined as a proposition that is trivially true.

**axiomatization**
\[
\text{false} :: \ o \ \text{(⊥)} \ \text{where}
\text{falseE [elim]}: \ \bot \Rightarrow A
\]

**definition**
\[
\text{true} :: \ o \ \text{(⊤)} \ \text{where}
\top \equiv \bot \Rightarrow \bot
\]
**CHAPTER 2. THE ISABELLE/ISAR FRAMEWORK**

```plaintext

**theorem** trueI [intro]: ⊤

**unfolding** true_def ..

```

Now negation represents an implication towards absurdity:

**definition**

`not :: o ⇒ o (¬ _ [40] 40) where
¬ A ≡ A → ⊥`

```plaintext

**theorem** notI [intro]:

**assumes** A ⇒ ⊥

**shows** ¬ A

**unfolding** not_def

**proof**

assume A

then show ⊥ by (rule ‘A ⇒ ⊥’)

qed

```

```plaintext

**theorem** notE [elim]:

**assumes** ¬ A and A

**shows** B

**proof**

from ‘¬ A’ have A ⇒ ⊥ unfolding not_def .

from ‘A ⇒ ⊥’ and ‘A’ have ⊥ ..

then show B ..

qed

```

### 2.3.4 Classical logic

Subsequently we state the principle of classical contradiction as a local assumption. Thus we refrain from forcing the object-logic into the classical perspective. Within that context, we may derive well-known consequences of the classical principle.

```plaintext

**locale** classical =

**assumes** classical: (¬ C ⇒ C) ⇒ C

**begin**

**theorem** double_negation:

**assumes** ¬ ¬ C

**shows** C

**proof** (rule classical)

assume ¬ C

with ‘¬ ¬ C’ show C ..

```

```
theorem tertium_non_datur: C ∨ ¬ C
proof (rule double_negation)
  show ¬ ¬ (C ∨ ¬ C)
proof
  assume ¬ (C ∨ ¬ C)
  have ¬ C
proof
    assume C then have C ∨ ¬ C ..
    with '¬ (C ∨ ¬ C)' show ⊥ ..
  qed
  then have C ∨ ¬ C ..
  with '¬ (C ∨ ¬ C)' show ⊥ ..
  qed
qed

These examples illustrate both classical reasoning and non-trivial propositional proofs in general. All three rules characterize classical logic independently, but the original rule is already the most convenient to use, because it leaves the conclusion unchanged. Note that (¬ C ⇒ C) ⇒ C fits again into our format for eliminations, despite the additional twist that the context refers to the main conclusion. So we may write classical as the Isar statement “obtains ¬ thesis”. This also explains nicely how classical reasoning really works: whatever the main thesis might be, we may always assume its negation!

end

2.3.5 Quantifiers

Representing quantifiers is easy, thanks to the higher-order nature of the underlying framework. According to the well-known technique introduced by Church [9], quantifiers are operators on predicates, which are syntactically represented as λ-terms of type i ⇒ o. Binder notation turns All (λx. B x) into ∀ x. B x etc.

axiomatization
  All :: (i ⇒ o) ⇒ o (binder ∀ 10) where
  allI [intro]: (\A x. B x) ⇒ ∀ x. B x and
  allD [dest]: (∀ x. B x) ⇒ B a

axiomatization
  Ex :: (i ⇒ o) ⇒ o (binder ∃ 10) where
exI \ [\text{intro}]: B a \implies (\exists x. B x) \text{ and}

exE \ [\text{elim}]: (\exists x. B x) \implies (\forall x. B x \implies C) \implies C

The statement of \text{exE} corresponds to “\text{assumes } \exists x. B x \text{ obtains } x \text{ where } B x” in Isar. In the subsequent example we illustrate quantifier reasoning involving all four rules:

\text{theorem}

\text{assumes } \exists x. \forall y. R x y

\text{shows } \forall y. \exists x. R x y

\text{proof}

\text{ obtain } x \text{ where } \forall y. R x y \text{ using } \exists x. \forall y. R x y \text{ .. } \begin{array}{l} \quad \exists \text{ elimination} \\
\text{fix } y \text{ have } R x y \text{ using } \forall y. R x y \text{ .. } \begin{array}{l} \quad \forall \text{ destruction} \\
\text{then show } \exists x. R x y \text{ .. } \begin{array}{l} \quad \exists \text{ introduction} \end{array} \\
\text{qed}
\end{array}
\end{array}

2.3.6 Canonical reasoning patterns

The main rules of first-order predicate logic from §2.3.3 and §2.3.5 can now be summarized as follows, using the native Isar statement format of §2.2.2.

\text{impI}: \text{assumes } A \implies B \text{ shows } A \longrightarrow B

\text{impD}: \text{assumes } A \longrightarrow B \text{ and } A \text{ shows } B

\text{disjI}_1: \text{assumes } A \text{ shows } A \lor B

\text{disjI}_2: \text{assumes } B \text{ shows } A \lor B

\text{disjE}: \text{assumes } A \lor B \text{ obtains } A \mid B

\text{conjI}: \text{assumes } A \text{ and } B \text{ shows } A \land B

\text{conjE}: \text{assumes } A \land B \text{ obtains } A \text{ and } B

\text{falseE}: \text{assumes } \bot \text{ shows } A

\text{trueI}: \text{shows } \top

\text{notI}: \text{assumes } A \implies \bot \text{ shows } \neg A

\text{notE}: \text{assumes } \neg A \text{ and } A \text{ shows } B

\text{allI}: \text{assumes } \forall x. B x \text{ shows } \forall x. B x

\text{allE}: \text{assumes } \forall x. B x \text{ shows } B a

\text{exI}: \text{assumes } B a \text{ shows } \exists x. B x

\text{exE}: \text{assumes } \exists x. B x \text{ obtains } a \text{ where } B a

This essentially provides a declarative reading of Pure rules as Isar reasoning patterns: the rule statements tells how a canonical proof outline shall look like. Since the above rules have already been declared as \text{intro, elim, dest} — each according to its particular shape — we can immediately write Isar proof texts as follows:
have \( A \rightarrow B \)  
proof  
assume \( A \)  
show \( B \) \( \langle \text{proof} \rangle \)  
qed

have \( A \rightarrow B \) and \( A \) \( \langle \text{proof} \rangle \)  
then have \( B \) ..

have \( A \) \( \langle \text{proof} \rangle \)  
then have \( A \lor B \) ..

have \( B \) \( \langle \text{proof} \rangle \)  
then have \( A \lor B \) ..

have \( A \land B \) \( \langle \text{proof} \rangle \)  
then have \( A \land B \) ..

have \( A \land B \) \( \langle \text{proof} \rangle \)  
then obtain \( A \) and \( B \) ..

have \( \bot \) \( \langle \text{proof} \rangle \)  
then have \( A \) ..

have \( \top \) ..

have \( \neg A \)  
proof  
assume \( A \)  
then show \( \bot \) \( \langle \text{proof} \rangle \)  
qed

have \( \neg A \) and \( A \) \( \langle \text{proof} \rangle \)  
then have \( B \) ..

have \( \forall x. \ B \ x \)  
proof  
fix \( x \)  
show \( B \ x \) \( \langle \text{proof} \rangle \)  
qed

have \( \forall x. \ B \ x \) \( \langle \text{proof} \rangle \)  
then have \( B \ a \) ..

have \( \exists x. \ B \ x \)  
proof  
show \( B \ a \) \( \langle \text{proof} \rangle \)  
qed

have \( \exists x. \ B \ x \) \( \langle \text{proof} \rangle \)  
then obtain \( a \) where \( B \ a \) ..

Of course, these proofs are merely examples. As sketched in §2.2.3, there is a fair amount of flexibility in expressing Pure deductions in Isar. Here the
user is asked to express himself adequately, aiming at proof texts of literary quality.

end
Part II

General Language Elements
Chapter 3

Outer syntax — the theory language

The rather generic framework of Isabelle/Isar syntax emerges from three main syntactic categories: commands of the top-level Isar engine (covering theory and proof elements), methods for general goal refinements (analogous to traditional “tactics”), and attributes for operations on facts (within a certain context). Subsequently we give a reference of basic syntactic entities underlying Isabelle/Isar syntax in a bottom-up manner. Concrete theory and proof language elements will be introduced later on.

In order to get started with writing well-formed Isabelle/Isar documents, the most important aspect to be noted is the difference of inner versus outer syntax. Inner syntax is that of Isabelle types and terms of the logic, while outer syntax is that of Isabelle/Isar theory sources (specifications and proofs). As a general rule, inner syntax entities may occur only as atomic entities within outer syntax. For example, the string "x + y" and identifier z are legal term specifications within a theory, while x + y without quotes is not. Printed theory documents usually omit quotes to gain readability (this is a matter of \isabellestyle macro setup, say via \isabellestyle, see also [48]). Experienced users of Isabelle/Isar may easily reconstruct the lost technical information, while mere readers need not care about quotes at all.

Isabelle/Isar input may contain any number of input termination characters “;” (semicolon) to separate commands explicitly. This is particularly useful in interactive shell sessions to make clear where the current command is intended to end. Otherwise, the interpreter loop will continue to issue a secondary prompt “#” until an end-of-command is clearly recognized from the input syntax, e.g., encounter of the next command keyword.

More advanced interfaces such as Proof General [2] do not require explicit semicolons, the amount of input text is determined automatically by inspecting the present content of the Emacs text buffer. In the printed presentation of Isabelle/Isar documents semicolons are omitted altogether for readability.
Proof General requires certain syntax classification tables in order to achieve properly synchronized interaction with the Isabelle/Isar process. These tables need to be consistent with the Isabelle version and particular logic image to be used in a running session (common object-logics may well change the outer syntax). The standard setup should work correctly with any of the "official" logic images derived from Isabelle/HOL (including HOLCF etc.). Users of alternative logics may need to tell Proof General explicitly, e.g. by giving an option -k ZF (in conjunction with -l ZF, to specify the default logic image). Note that option -L does both of this at the same time.

3.1 Lexical matters

The outer lexical syntax consists of three main categories of syntax tokens:

1. **major keywords** — the command names that are available in the present logic session;
2. **minor keywords** — additional literal tokens required by the syntax of commands;
3. **named tokens** — various categories of identifiers etc.

Major keywords and minor keywords are guaranteed to be disjoint. This helps user-interfaces to determine the overall structure of a theory text, without knowing the full details of command syntax. Internally, there is some additional information about the kind of major keywords, which approximates the command type (theory command, proof command etc.).

Keywords override named tokens. For example, the presence of a command called `term` inhibits the identifier `term`, but the string "term" can be used instead. By convention, the outer syntax always allows quoted strings in addition to identifiers, wherever a named entity is expected.

When tokenizing a given input sequence, the lexer repeatedly takes the longest prefix of the input that forms a valid token. Spaces, tabs, newlines and formfeeds between tokens serve as explicit separators.

The categories for named tokens are defined once and for all as follows.

\[
\begin{align*}
\text{ident} & = \text{letter quasiletter}^* \\
\text{longident} & = \text{ident} (\cdot \text{ident})^+ \\
\text{symident} & = \text{sym}^+ \mid \text{\backslash ident}
\end{align*}
\]
A \texttt{var} or \texttt{typevar} describes an unknown, which is internally a pair of base name and index (ML type \texttt{indexname}). These components are either separated by a dot as in \texttt{?x.1} or \texttt{?x?7.3} or run together as in \texttt{?x1}. The latter form is possible if the base name does not end with digits. If the index is 0, it may be dropped altogether: \texttt{?x} and \texttt{?x0} and \texttt{?x.0} all refer to the same unknown, with basename \texttt{x} and index 0.

The syntax of \texttt{string} admits any characters, including newlines; \texttt{""} (double-quote) and \texttt{"\textbackslash"} (backslash) need to be escaped by a backslash; arbitrary character codes may be specified as \texttt{\textbackslash ddd}, with three decimal digits. Alternative strings according to \texttt{altstring} are analogous, using single back-quotes instead.

The body of \texttt{verbatim} may consist of any text not containing \texttt{"\textbackslash\textbackslash"}; this allows convenient inclusion of quotes without further escapes. There is no way to escape \texttt{\textbackslash\textbackslash"}. If the quoted text is \LaTeX{} source, one may usually add some blank or comment to avoid the critical character sequence.
Source comments take the form (* \ldots *) and may be nested, although the user-interface might prevent this. Note that this form indicates source comments only, which are stripped after lexical analysis of the input. The Isar syntax also provides proper document comments that are considered as part of the text (see §3.2.3).

Common mathematical symbols such as $\forall$ are represented in Isabelle as $\backslash$forall. There are infinitely many Isabelle symbols like this, although proper presentation is left to front-end tools such as \LaTeX, Proof General, or Isabelle/jEdit. A list of predefined Isabelle symbols that work well with these tools is given in appendix B. Note that $\backslash$lambda does not belong to the letter category, since it is already used differently in the Pure term language.

### 3.2 Common syntax entities

We now introduce several basic syntactic entities, such as names, terms, and theorem specifications, which are factored out of the actual Isar language elements to be described later.

#### 3.2.1 Names

Entity \textit{name} usually refers to any name of types, constants, theorems etc. that are to be declared or defined (so qualified identifiers are excluded here). Quoted strings provide an escape for non-identifier names or those ruled out by outer syntax keywords (e.g. quoted "let"). Already existing objects are usually referenced by \textit{nameref}.

\begin{center}
\begin{tikzpicture}
  \node (name) {name};
  \node (ident) [below left of=name] {ident};
  \node (symident) [below of=ident] {symident};
  \node (string) [below of=symident] {string};
  \node (nat) [below of=string] {nat};
  \node (parname) [below of=ident] {\langle \rangle};
  \draw (name) -- (ident);
  \draw (ident) -- (symident);
  \draw (symident) -- (string);
  \draw (string) -- (nat);
  \draw (name) -- (parname);
\end{tikzpicture}
\end{center}
3.2.2 Numbers

The outer lexical syntax (§3.1) admits natural numbers and floating point numbers. These are combined as int and real as follows.

\[\text{int} \quad \text{nat} \quad \text{real} \quad \text{float} \quad \text{int}\]

Note that there is an overlap with the category name, which also includes nat.

3.2.3 Comments

Large chunks of plain text are usually given verbatim, i.e. enclosed in \{\* \ldots \*\}. For convenience, any of the smaller text units conforming to nameref are admitted as well. A marginal comment is of the form -- text. Any number of these may occur within Isabelle/Isar commands.

\[\text{text} \quad \text{verbatim} \quad \text{nameref}\]
3.2.4 Type classes, sorts and arities

Classes are specified by plain names. Sorts have a very simple inner syntax, which is either a single class name $c$ or a list $\{c_1, \ldots, c_n\}$ referring to the intersection of these classes. The syntax of type arities is given directly at the outer level.

3.2.5 Types and terms

The actual inner Isabelle syntax, that of types and terms of the logic, is far too sophisticated in order to be modelled explicitly at the outer theory level. Basically, any such entity has to be quoted to turn it into a single token (the parsing and type-checking is performed internally later). For convenience, a slightly more liberal convention is adopted: quotes may be omitted for any
type or term that is already atomic at the outer level. For example, one may just write \(x\) instead of quoted "\(x\)". Note that symbolic identifiers (e.g. ++ or \(\forall\)) are available as well, provided these have not been superseded by commands or other keywords already (such as = or +).

Positional instantiations are indicated by giving a sequence of terms, or the placeholder "\(_\)" (underscore), which means to skip a position.

Type declarations and definitions usually refer to typespec on the left-hand side. This models basic type constructor application at the outer syntax level. Note that only plain postfix notation is available here, but no infixes.
3.2.6 Term patterns and declarations

Wherever explicit propositions (or term fragments) occur in a proof text, casual binding of schematic term variables may be given specified via patterns of the form “(is \(p_1 \ldots p_n\))”. This works both for \textit{term} and \textit{prop}.

\textit{term\_pat}

\textit{prop\_pat}
Declarations of local variables $x :: \tau$ and logical propositions $a : \varphi$ represent different views on the same principle of introducing a local scope. In practice, one may usually omit the typing of $\textit{vars}$ (due to type-inference), and the naming of propositions (due to implicit references of current facts). In any case, Isar proof elements usually admit to introduce multiple such items simultaneously.

The treatment of multiple declarations corresponds to the complementary focus of $\textit{vars}$ versus $\textit{props}$. In “$x_1 \ldots x_n :: \tau$” the typing refers to all variables, while in $a : \varphi_1 \ldots \varphi_n$ the naming refers to all propositions collectively. Isar language elements that refer to $\textit{vars}$ or $\textit{props}$ typically admit separate typings or namings via another level of iteration, with explicit and separators; e.g., see \texttt{fix} and \texttt{assume} in §6.2.1.

### 3.2.7 Attributes and theorems

Attributes have their own “semi-inner” syntax, in the sense that input conforming to $\texttt{args}$ below is parsed by the attribute a second time. The attribute argument specifications may be any sequence of atomic entities (identifiers, strings etc.), or properly bracketed argument lists. Below \texttt{atom} refers to any atomic entity, including any \texttt{keyword} conforming to \texttt{symident}.
Theorem specifications come in several flavors: `axmdecl` and `thmdecl` usually refer to axioms, assumptions or results of goal statements, while `thmdef` collects lists of existing theorems. Existing theorems are given by `thmref` and `thmrefs`, the former requires an actual singleton result.
CHAPTER 3. OUTER SYNTAX — THE THEORY LANGUAGE

There are three forms of theorem references:

1. named facts \( a \),
2. selections from named facts \( a(i) \) or \( a(j-k) \),
3. literal fact propositions using \( \text{altstring} \) syntax ‘\( \varphi \)’ (see also method \texttt{fact}).

Any kind of theorem specification may include lists of attributes both on the left and right hand sides; attributes are applied to any immediately preceding fact. If names are omitted, the theorems are not stored within the theorem database of the theory or proof context, but any given attributes are applied nonetheless.

An extra pair of brackets around attributes (like “[\texttt{simproc} \( a \)]”) abbreviates a theorem reference involving an internal dummy fact, which will be ignored later on. So only the effect of the attribute on the background context will persist. This form of in-place declarations is particularly useful with commands like \texttt{declare} and \texttt{using}.

\texttt{axmdecl}

\texttt{thmdecl}

\texttt{thmdef}

\texttt{thmref}
**CHAPTER 3. OUTER SYNTAX — THE THEORY LANGUAGE**

\[ \text{thmrefs} \]

\[ \text{thmbind} \]

\[ \text{selection} \]
Document preparation

Isabelle/Isar provides a simple document preparation system based on regular PDF-LATEX technology, with full support for hyper-links and bookmarks. Thus the results are well suited for WWW browsing and as printed copies.

Isabelle generates LATEX output while running a logic session (see also [48]). Getting started with a working configuration for common situations is quite easy by using the Isabelle mkdir and make tools. First invoke

```
isabelle mkdir Foo
```

to initialize a separate directory for session Foo (it is safe to experiment, since isabelle mkdir never overwrites existing files). Ensure that Foo/ROOT.ML holds ML commands to load all theories required for this session; furthermore Foo/document/root.tex should include any special LATEX macro packages required for your document (the default is usually sufficient as a start).

The session is controlled by a separate IsaMakefile (with crude source dependencies by default). This file is located one level up from the Foo directory location. Now invoke

```
isabelle make Foo
```

to run the Foo session, with browser information and document preparation enabled. Unless any errors are reported by Isabelle or LATEX, the output will appear inside the directory defined by the ISABELLE_BROWSER_INFO setting (as reported by the batch job in verbose mode).

You may also consider to tune the usedir options in IsaMakefile, for example to switch the output format between pdf and dvi, or activate the -D option to retain a second copy of the generated LATEX sources (for manual inspection or separate runs of latex).

See The Isabelle System Manual [48] for further details on Isabelle logic sessions and theory presentation. The Isabelle/HOL tutorial [25] also covers theory presentation to some extent.
4.1 Markup commands

```
header : toplevel → toplevel
chapter : local_theory → local_theory
section : local_theory → local_theory
subsection : local_theory → local_theory
subsubsection : local_theory → local_theory
  text : local_theory → local_theory
  text_raw : local_theory → local_theory
    sect : proof → proof
    subsect : proof → proof
    subsubsect : proof → proof
    txt : proof → proof
    txt_raw : proof → proof
```

Markup commands provide a structured way to insert text into the document generated from a theory. Each markup command takes a single text argument, which is passed as argument to a corresponding \LaTeX macro. The default macros provided by "\~\~/lib/texinputs/isabelle.sty can be redefined according to the needs of the underlying document and \LaTeX styles.

Note that formal comments (§3.2.3) are similar to markup commands, but have a different status within Isabelle/Isar syntax.
header provides plain text markup just preceding the formal beginning of a theory. The corresponding \isamarkupheader \isamarkupheader, which acts like section by default.

chapter, section, subsection, and subsubsection mark chapter and section headings within the main theory body or local theory targets. The corresponding \isamarkupchapter, \isamarkupsection, \isamarkupsubsection etc.

sect, subsect, and subsubsect mark section headings within proofs. The corresponding \isamarkupsect, \isamarkupsubsect, etc.

text and txt specify paragraphs of plain text. This corresponds to a \isamarkuptext \isamarkuptext ... \end{isamarkuptext} etc.

text_raw and txt_raw insert \isamarkupsource into the output, without additional markup. Thus the full range of document manipulations becomes available, at the risk of messing up document output.

Except for text_raw and txt_raw, the text passed to any of the above markup commands may refer to formal entities via document antiquotations, see also §4.2. These are interpreted in the present theory or proof context, or the named target.

The proof markup commands closely resemble those for theory specifications, but have a different formal status and produce different \isamarkupsource macros. The default definitions coincide for analogous commands such as section and sect.
4.2 Document Antiquotations

The overall content of an Isabelle/Isar theory may alternate between formal and informal text. The main body consists of formal specification and proof commands, interspersed with markup commands (§4.1) or document comments (§3.2.3). The argument of markup commands quotes informal text to be printed in the resulting document, but may again refer to formal entities via document antiquotations.

For example, embedding of “@{term [show_types] f x = a + x}” within a text block makes \((f::'a ⇒ 'a) (x::'a) = (a::'a) + x\) appear in the final \(\LaTeX\) document.

Antiquotations usually spare the author tedious typing of logical entities in full detail. Even more importantly, some degree of consistency-checking between the main body of formal text and its informal explanation is achieved, since terms and types appearing in antiquotations are checked within the current theory or proof context.
antiquotation

antiquotation

theory [options] [name]

thm [options] [styles] [thmrefs]

lemma [options] [prop] [by] [method]

prop [options] [styles] [prop]

term [options] [styles] [term]

value [options] [styles] [term]

term_type [options] [styles] [term]

typeof [options] [styles] [term]

const [options] [term]

abbrev [options] [term]

typ [options] [type]

type [options] [name]

class [options] [name]

text [options] [name]
CHAPTER 4. DOCUMENT PREPARATION

antiquotation

goals options

subgoals options

prf options thmrefs

full_prf options thmrefs

ML options name

ML_op options name

ML_type options name

ML_struct options name

file options name

options

[ ]

option

, 

option

name

name = name

styles

( style )

, 

style

name

Note that the syntax of antiquotations may not include source comments (* ... *) nor verbatim text {* ... *).

\{@\{theory A\}\} prints the name \(A\), which is guaranteed to refer to a valid ancestor theory in the current context.

\{@\{thm a_1 \ldots a_n\}\} prints theorems \(a_1 \ldots a_n\). Full fact expressions are allowed here, including attributes (§3.2.7).

\{@\{prop \varphi\}\} prints a well-typed proposition \(\varphi\).

\{@\{lemma \varphi \ by \ m\}\} proves a well-typed proposition \(\varphi\) by method \(m\) and prints the original \(\varphi\).

\{@\{term t\}\} prints a well-typed term \(t\).

\{@\{value t\}\} evaluates a term \(t\) and prints its result, see also \texttt{value}.

\{@\{term\_type t\}\} prints a well-typed term \(t\) annotated with its type.

\{@\{typeof t\}\} prints the type of a well-typed term \(t\).

\{@\{const c\}\} prints a logical or syntactic constant \(c\).

\{@\{abbrev c \, x_1 \ldots x_n\}\} prints a constant abbreviation \(c \, x_1 \ldots x_n \equiv \text{rhs}\) as defined in the current context.

\{@\{typ \tau\}\} prints a well-formed type \(\tau\).

\{@\{type \kappa\}\} prints a (logical or syntactic) type constructor \(\kappa\).

\{@\{class c\}\} prints a class \(c\).

\{@\{text s\}\} prints uninterpreted source text \(s\). This is particularly useful to print portions of text according to the Isabelle document style, without demanding well-formedness, e.g. small pieces of terms that should not be parsed or type-checked yet.
\@\{goals\} prints the current \textit{dynamic} goal state. This is mainly for support of tactic-emulation scripts within Isar. Presentation of goal states does not conform to the idea of human-readable proof documents!

When explaining proofs in detail it is usually better to spell out the reasoning via proper Isar proof commands, instead of peeking at the internal machine configuration.

\@\{subgoals\} is similar to \@\{goals\}, but does not print the main goal.

\@\{prf a_1 \ldots a_n\} prints the (compact) proof terms corresponding to the theorems \(a_1 \ldots a_n\). Note that this requires proof terms to be switched on for the current logic session.

\@\{full\_prf a_1 \ldots a_n\} is like \@\{prf a_1 \ldots a_n\}, but prints the full proof terms, i.e. also displays information omitted in the compact proof term, which is denoted by \texttt{"\_\_"} placeholders there.

\@\{ML s\}, \@\{ML\_op s\}, \@\{ML\_type s\}, and \@\{ML\_struct s\} check text \(s\) as ML value, infix operator, type, and structure, respectively. The source is printed verbatim.

\@\{file path\} checks that \texttt{path} refers to a file (or directory) and prints it verbatim.

### 4.2.1 Styled antiquotations

The antiquotations \texttt{thm}, \texttt{prop} and \texttt{term} admit an \textit{extra style} specification to modify the printed result. A style is specified by a name with a possibly empty number of arguments; multiple styles can be sequenced with commas. The following standard styles are available:

- \texttt{lhs} extracts the first argument of any application form with at least two arguments — typically meta-level or object-level equality, or any other binary relation.

- \texttt{rhs} is like \texttt{lhs}, but extracts the second argument.

- \texttt{concl} extracts the conclusion \(C\) from a rule in Horn-clause normal form \(A_1 \implies \ldots A_n \implies C\).

- \texttt{prem n} extract premise number \(n\) from from a rule in Horn-clause normal form \(A_1 \implies \ldots A_n \implies C\)
4.2.2 General options

The following options are available to tune the printed output of antiquotations. Note that many of these coincide with global ML flags of the same names.

- \textit{show_types} = bool and \textit{show_sorts} = bool control printing of explicit type and sort constraints.
- \textit{show_structs} = bool controls printing of implicit structures.
- \textit{show_abbrs} = bool controls folding of abbreviations.
- \textit{names_long} = bool forces names of types and constants etc. to be printed in their fully qualified internal form.
- \textit{names_short} = bool forces names of types and constants etc. to be printed unqualified. Note that internalizing the output again in the current context may well yield a different result.
- \textit{names_unique} = bool determines whether the printed version of qualified names should be made sufficiently long to avoid overlap with names declared further back. Set to \textit{false} for more concise output.
- \textit{eta_contract} = bool prints terms in \(\eta\)-contracted form.
- \textit{display} = bool indicates if the text is to be output as multi-line “display material”, rather than a small piece of text without line breaks (which is the default).
  In this mode the embedded entities are printed in the same style as the main theory text.
- \textit{break} = bool controls line breaks in non-display material.
- \textit{quotes} = bool indicates if the output should be enclosed in double quotes.
- \textit{mode} = name adds \textit{name} to the print mode to be used for presentation. Note that the standard setup for \LaTeX output is already present by default, including the modes \textit{latex} and \textit{xsymbols}.
- \textit{margin} = nat and \textit{indent} = nat change the margin or indentation for pretty printing of display material.
- \textit{goals.limit} = nat determines the maximum number of goals to be printed (for goal-based antiquotation).
source = bool prints the original source text of the antiquotation arguments, rather than its internal representation. Note that formal checking of \texttt{thm}, \texttt{term}, etc. is still enabled; use the \texttt{text} antiquotation for unchecked output.

Regular \texttt{term} and \texttt{typ} antiquotations with source = false involve a full round-trip from the original source to an internalized logical entity back to a source form, according to the syntax of the current context. Thus the printed output is not under direct control of the author, it may even fluctuate a bit as the underlying theory is changed later on.

In contrast, source = true admits direct printing of the given source text, with the desirable well-formedness check in the background, but without modification of the printed text.

For boolean flags, “name = true” may be abbreviated as “name”. All of the above flags are disabled by default, unless changed from ML, say in the \texttt{ROOT.ML} of the logic session.

### 4.3 Markup via command tags

Each Isabelle/Isar command may be decorated by additional presentation tags, to indicate some modification in the way it is printed in the document.

```
tag

% ident string
```

Some tags are pre-declared for certain classes of commands, serving as default markup if no tags are given in the text:

- \texttt{theory} theory begin/end
- \texttt{proof} all proof commands
- \texttt{ML} all commands involving ML code
CHAPTER 4. DOCUMENT PREPARATION

The Isabelle document preparation system [48] allows tagged command regions to be presented specifically, e.g. to fold proof texts, or drop parts of the text completely.

For example “by \%invisible auto” causes that piece of proof to be treated as invisible instead of proof (the default), which may be shown or hidden depending on the document setup. In contrast, “by \%visible auto” forces this text to be shown invariably.

Explicit tag specifications within a proof apply to all subsequent commands of the same level of nesting. For example, “proof \%visible \ldots qed” forces the whole sub-proof to be typeset as visible (unless some of its parts are tagged differently).

Command tags merely produce certain markup environments for typesetting. The meaning of these is determined by \LaTeX macros, as defined in ``/lib/texinputs/isabelle.sty or by the document author. The Isabelle document preparation tools also provide some high-level options to specify the meaning of arbitrary tags to “keep”, “drop”, or “fold” the corresponding parts of the text. Logic sessions may also specify “document versions”, where given tags are interpreted in some particular way. Again see [48] for further details.

4.4 Railroad diagrams

rail : antiquotation

\begin{rail}
  |rule|
  |\string|
\end{rail}

The rail antiquotation allows to include syntax diagrams into Isabelle documents. \LaTeX requires the style file ``/lib/texinputs/pdfsetup.sty, which can be used via \usepackage{pdfsetup} in root.tex, for example.

The rail specification language is quoted here as Isabelle string; it has its own grammar given below.
The lexical syntax of *identifier* coincides with that of *ident* in regular Isabelle syntax, but *string* uses single quotes instead of double quotes of the standard *string* category, to avoid extra escapes.

Each *rule* defines a formal language (with optional name), using a notation that is similar to EBNF or regular expressions with recursion. The meaning and visual appearance of these rail language elements is illustrated by the following representative examples.


- Empty ()

- Nonterminal A

- Nonterminal via Isabelle antiquotation @{syntax method}

- Terminal 'xyz'

- Terminal in keyword style @'xyz'

- Terminal via Isabelle antiquotation @@{method rule}

- Concatenation A B C

- Linebreak \ inside concatenation¹ A B C \ D E F

¹Strictly speaking, this is only a single backslash, but the enclosing string syntax requires a second one for escaping.
• Variants A | B | C

• Option A ?

• Repetition A *

• Repetition with separator A * sep

• Strict repetition A +
• Strict repetition with separator $A + \text{sep}$

4.5 Draft presentation

\begin{align*}
\text{display_drafts}^* & : \text{any} \rightarrow \\
\text{print_drafts}^* & : \text{any} \rightarrow
\end{align*}

\textbf{display_drafts} paths and \textbf{print_drafts} paths perform simple output of a given list of raw source files. Only those symbols that do not require additional \LaTeXX packages are displayed properly, everything else is left verbatim.
Chapter 5

Specifications

The Isabelle/Isar theory format integrates specifications and proofs, supporting interactive development with unlimited undo operation. There is an integrated document preparation system (see chapter 4), for typesetting formal developments together with informal text. The resulting hyper-linked PDF documents can be used both for WWW presentation and printed copies.

The Isar proof language (see chapter 6) is embedded into the theory language as a proper sub-language. Proof mode is entered by stating some theorem or lemma at the theory level, and left again with the final conclusion (e.g. via qed). Some theory specification mechanisms also require a proof, such as typedef in HOL, which demands non-emptiness of the representing sets.

5.1 Defining theories

theory : toplevel → theory  
end : theory → toplevel

Isabelle/Isar theories are defined via theory files, which may contain both specifications and proofs; occasionally definitional mechanisms also require some explicit proof. The theory body may be sub-structured by means of local theory targets, such as locale and class.

The first proper command of a theory is theory, which indicates imports of previous theories and optional dependencies on other source files (usually in ML). Just preceding the initial theory command there may be an optional header declaration, which is only relevant to document preparation; see also the other section markup commands in §4.1.

A theory is concluded by a final end command, one that does not belong to a local theory target. No further commands may follow such a global end, although some user-interfaces might pretend that trailing input is admissible.
The optional **keywords** specification declares outer syntax (chapter 3) that is introduced in this theory later on (rare in end-user applications). Both minor keywords and major keywords of the Isar command language need to be specified, in order to make parsing of proof documents
work properly. Command keywords need to be classified according to their structural role in the formal text. Examples may be seen in Isabelle/HOL sources itself, such as keywords "typedef" ::thy_goal or keywords "datatype" :: thy_decl for theory-level declarations with and without proof, respectively. Additional tags provide defaults for document preparation (§4.3).

The optional uses specification declares additional dependencies on external files (notably ML sources). Files will be loaded immediately (as ML), unless the name is parenthesized. The latter case records a dependency that needs to be resolved later in the text, usually via explicit use for ML files; other file formats require specific load commands defined by the corresponding tools or packages.

end concludes the current theory definition. Note that some other commands, e.g. local theory targets locale or class may involve a begin that needs to be matched by end, according to the usual rules for nested blocks.

5.2 Local theory targets

context : theory → local_theory
end : local_theory → theory

A local theory target is a context managed separately within the enclosing theory. Contexts may introduce parameters (fixed variables) and assumptions (hypotheses). Definitions and theorems depending on the context may be added incrementally later on.

Named contexts refer to locales (cf. §5.6) or type classes (cf. §5.7); the name “−” signifies the global theory context.

Unnamed contexts may introduce additional parameters and assumptions, and results produced in the context are generalized accordingly. Such auxiliary contexts may be nested within other targets, like locale, class, instantiation, overloading.

\[ \text{context} \quad \text{nameref} \quad \text{begin} \]

\[ \text{context} \quad \text{includes} \quad \text{context_elem} \quad \text{begin} \]
target

\begin{center}
\begin{tabular}{ccc}
\textbf{context} & \textbf{in} & \textbf{nameref} \\
\hline
\textbf{c} & \textbf{\texttt{begin}} & opens a named context, by recommencing an existing \\
locale or class \texttt{c}. Note that locale and class definitions allow to in-
clude the \texttt{begin} keyword as well, in order to continue the local theory 
immediately after the initial specification.
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{ccc}
\textbf{context} & \textbf{bundles elements} & \textbf{begin} \\
\hline
opens an unnamed context, by extending the enclosing global or local theory target by the given declaration 
bundles (§5.3) and context elements (\texttt{fixes, assumes} etc.). This means 
any results stemming from definitions and proofs in the extended context will be exported into the enclosing target by lifting over extra 
parameters and premises.
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{ccc}
\textbf{end} & concludes the current local theory, according to the nesting of contexts. 
& Note that a global \texttt{end} has a different meaning: it concludes the theory 
itself (§5.1).
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{ccc}
\texttt{(in c)} & given after any local theory command specifies an immediate target, 
e.g. \texttt{"definition (in c) \ldots"} or \texttt{"theorem (in c) \ldots"}. This works both 
in a local or global theory context; the current target context will be 
suspended for this command only. Note that \texttt{"(in −)"} will always 
produce a global result independently of the current target context.
\end{tabular}
\end{center}

The exact meaning of results produced within a local theory context de-
pends on the underlying target infrastructure (locale, type class etc.). The 
general idea is as follows, considering a context named \texttt{c} with parameter \texttt{x} 
and assumption \texttt{A[x]}.

Definitions are exported by introducing a global version with additional arg-
ments; a syntactic abbreviation links the long form with the abstract version 
of the target context. For example, \texttt{a ≡ t[x]} becomes \texttt{c.a ?x ≡ t[?x]} at the 
theory level (for arbitrary \texttt{?x}), together with a local abbreviation \texttt{c ≡ c.a x} 
in the target context (for the fixed parameter \texttt{x}).

Theorems are exported by discharging the assumptions and generalizing the 
parameters of the context. For example, \texttt{a: B[x]} becomes \texttt{c.a: A[?x] ⇒} 
\texttt{B[?x]}, again for arbitrary \texttt{?x}.

The Isabelle/HOL library contains numerous applications of locales and 
classes, e.g. see \texttt{"/src/HOL/Algebra}. An example for an unnamed auxiliary 
contexts is given in \texttt{"/src/HOL/Isar_Examples/Group_Context.thy}. 
5.3 Bundled declarations

\begin{align*}
\text{bundle} &: \text{local\_theory} \to \text{local\_theory} \\
\text{print\_bundles}^* &: \text{context} \to \\
\text{include} &: \text{proof\(\text{state}\)} \to \text{proof\(\text{state}\)} \\
\text{including} &: \text{proof\(\text{prove}\)} \to \text{proof\(\text{prove}\)} \\
\text{includes} &: \text{syntax}
\end{align*}

The outer syntax of fact expressions (§3.2.7) involves theorems and attributes, which are evaluated in the context and applied to it. Attributes may declare theorems to the context, as in \texttt{this\_rule [intro] that\_rule [elim]} for example. Configuration options (§9.1) are special declaration attributes that operate on the context without a theorem, as in \texttt{[[show\_types = false]]} for example.

Expressions of this form may be defined as \textit{bundled declarations} in the context, and included in other situations later on. Including declaration bundles augments a local context casually without logical dependencies, which is in contrast to locales and locale interpretation (§5.6).
bundle $b = decls$ defines a bundle of declarations in the current context. The RHS is similar to the one of the declare command. Bundles defined in local theory targets are subject to transformations via morphisms, when moved into different application contexts; this works analogously to any other local theory specification.

print_bundles prints the named bundles that are available in the current context.

include $b_1 \ldots b_n$ includes the declarations from the given bundles into the current proof body context. This is analogous to note (§6.2.3) with the expanded bundles.

including is similar to include, but works in proof refinement (backward mode). This is analogous to using (§6.2.3) with the expanded bundles.

includes $b_1 \ldots b_n$ is similar to include, but works in situations where a specification context is constructed, notably for context and long statements of theorem etc.

Here is an artificial example of bundling various configuration options:

bundle trace $= [\{ simp_trace, blast_trace, linarith_trace, metis_trace, smt_trace \}]$

lemma $x = x$
including trace by metis

5.4 Basic specification elements

axiomatization : $theory \rightarrow theory$ (axiomatic!)
definition : $local\_theory \rightarrow local\_theory$
defn : attribute
abbreviation : $local\_theory \rightarrow local\_theory$
print_abbrevs* : $context \rightarrow$

These specification mechanisms provide a slightly more abstract view than the underlying primitives of consts, defs (see §5.10.3), and axioms (see §5.11). In particular, type-inference is commonly available, and result names need not be given.
definition

abbreviation

fixes

specs
axiomatization \( c_1 \ldots c_m \) \textbf{where} \( \varphi_1 \ldots \varphi_n \) introduces several constants simultaneously and states axiomatic properties for these. The constants are marked as being specified once and for all, which prevents additional specifications being issued later on.

Note that axiomatic specifications are only appropriate when declaring a new logical system; axiomatic specifications are restricted to global theory contexts. Normal applications should only use definitional mechanisms!

definition \( c \) \textbf{where} \( eq \) produces an internal definition \( c \equiv t \) according to the specification given as \( eq \), which is then turned into a proven fact. The given proposition may deviate from internal meta-level equality according to the rewrite rules declared as \textit{defn} by the object-logic. This usually covers object-level equality \( x = y \) and equivalence \( A \leftrightarrow B \). End-users normally need not change the \textit{defn} setup.

Definitions may be presented with explicit arguments on the LHS, as well as additional conditions, e.g. \( f x y = t \) instead of \( f \equiv \lambda x y. \ t \) and \( y \neq 0 \implies g x y = u \) instead of an unrestricted \( g \equiv \lambda x y. \ u \).

abbreviation \( c \) \textbf{where} \( eq \) introduces a syntactic constant which is associated with a certain term according to the meta-level equality \( eq \).

Abbreviations participate in the usual type-inference process, but are expanded before the logic ever sees them. Pretty printing of terms involves higher-order rewriting with rules stemming from reverted abbreviations. This needs some care to avoid overlapping or looping syntactic replacements!

The optional \textit{mode} specification restricts output to a particular print mode; using “\textit{input}” here achieves the effect of one-way abbreviations. The mode may also include an “\textit{output}” qualifier that affects the concrete syntax declared for abbreviations, cf. \textit{syntax} in §7.5.

\texttt{print.abbrevs} prints all constant abbreviations of the current context.
5.5 Generic declarations

\[
\begin{align*}
\text{declaration} &: \, \text{local}\_\text{theory} \rightarrow \text{local}\_\text{theory} \\
\text{syntax\_declaration} &: \, \text{local}\_\text{theory} \rightarrow \text{local}\_\text{theory} \\
\text{declare} &: \, \text{local}\_\text{theory} \rightarrow \text{local}\_\text{theory}
\end{align*}
\]

Arbitrary operations on the background context may be wrapped-up as generic declaration elements. Since the underlying concept of local theories may be subject to later re-interpretation, there is an additional dependency on a morphism that tells the difference of the original declaration context wrt. the application context encountered later on. A fact declaration is an important special case: it consists of a theorem which is applied to the context by means of an attribute.

\[
\text{declaration} \
\begin{cases}
\text{syntax\_declaration} \rightarrow (\text{pervasive}) \\
\text{declare} \\
\text{text} \\
\text{target} \\
\text{declare} \\
\text{thmrefs} \\
\text{and} \\
\text{target}
\end{cases}
\]

\text{declaration} \, d \, \text{adds the declaration function} \, d \, \text{of ML type} \, \text{declaration}, \, \text{to the current local theory under construction. In later application contexts, the function is transformed according to the morphisms being involved in the interpretation hierarchy.}

If the (pervasive) option is given, the corresponding declaration is applied to all possible contexts involved, including the global background theory.

\text{syntax\_declaration} \, \text{is similar to} \, \text{declaration}, \, \text{but is meant to affect only “syntactic” tools by convention (such as notation and type-checking information).}
**CHAPTER 5. SPECIFICATIONS**

*declare thms* declares theorems to the current local theory context. No theorem binding is involved here, unlike *theorems* or *lemmas* (cf. §5.11), so *declare* only has the effect of applying attributes as included in the theorem specification.

### 5.6 Locales

Locales are parametric named local contexts, consisting of a list of declaration elements that are modeled after the Isar proof context commands (cf. §6.2.1).

#### 5.6.1 Locale expressions

A *locale expression* denotes a structured context composed of instances of existing locales. The context consists of a list of instances of declaration elements from the locales. Two locale instances are equal if they are of the same locale and the parameters are instantiated with equivalent terms. Declaration elements from equal instances are never repeated, thus avoiding duplicate declarations.

\[
\text{locale_expr}
\]

\[
\text{instance}
\]

\[
\text{specifier}
\]

\[
\text{name}
\]
A locale instance consists of a reference to a locale and either positional or named parameter instantiations. Identical instantiations (that is, those that instantiate a parameter by itself) may be omitted. The notation `\_` enables to omit the instantiation for a parameter inside a positional instantiation.

Terms in instantiations are from the context the locale expressions is declared in. Local names may be added to this context with the optional for clause. In addition, syntax declarations from one instance are effective when parsing subsequent instances of the same expression.

Instances have an optional qualifier which applies to names in declarations. Names include local definitions and theorem names. If present, the qualifier itself is either optional ("?"), which means that it may be omitted on input of the qualified name, or mandatory ("!"). If neither "?" nor "!" are present, the command’s default is used. For `interpretation` and `interpret` the default is “mandatory”, for `locale` and `sublocale` the default is “optional”.

### 5.6.2 Locale declarations

- `locale` : `theory → local_theory`
- `print_locale` : `context →`
- `print_locales` : `context →`
- `intro_locales` : `method`
- `unfold_locales` : `method`

```schematic
category 0
locale name = locale begin begin
```
locale $loc = \text{import} + \text{body}$ defines a new locale $loc$ as a context consisting of a certain view of existing locales ($\text{import}$) plus some additional elements ($\text{body}$). Both $\text{import}$ and $\text{body}$ are optional; the degenerate form $\text{locale} \ loc$ defines an empty locale, which may still be useful to collect declarations of facts later on. Type-inference on locale expressions automatically takes care of the most general typing that the combined context elements may acquire.

The $\text{import}$ consists of a structured locale expression; see §6.2.1 above. Its for clause defines the local parameters of the $\text{import}$. In addition, locale parameters whose instantance is omitted automatically extend the (possibly empty) for clause: they are inserted at its beginning. This means that these parameters may be referred to from within the expression and also in the subsequent context elements and provides a notational convenience for the inheritance of parameters in locale declarations.

The $\text{body}$ consists of context elements.

- \textbf{fixes} $x :: \tau$ ($mx$) declares a local parameter of type $\tau$ and mixfix annotation $mx$ (both are optional). The special syntax declaration “(structure)” means that $x$ may be referenced implicitly in this context.

- \textbf{constrains} $x :: \tau$ introduces a type constraint $\tau$ on the local parameter $x$. This element is deprecated. The type constraint should be introduced in the for clause or the relevant \textbf{fixes} element.

- \textbf{assumes} $a: \varphi_1 \ldots \varphi_n$ introduces local premises, similar to \textbf{assume} within a proof (cf. §6.2.1).

- \textbf{defines} $a: x \equiv t$ defines a previously declared parameter. This is similar to \textbf{def} within a proof (cf. §6.2.1), but \textbf{defines} takes an equational proposition instead of variable-term pair. The left-hand side of the equation may have additional arguments, e.g. “\textbf{defines} f x_1 \ldots x_n \equiv t”.

- \textbf{notes} $a = b_1 \ldots b_n$ reconsiders facts within a local context. Most notably, this may include arbitrary declarations in any attribute specifications included here, e.g. a local simp rule.

The initial $\text{import}$ specification of a locale expression maintains a dynamic relation to the locales being referenced (benefiting from any later fact declarations in the obvious manner).

Note that “(is $p_1 \ldots p_n$)” patterns given in the syntax of \textbf{assumes} and
defines above are illegal in locale definitions. In the long goal format of §6.2.4, term bindings may be included as expected, though.

Locale specifications are “closed up” by turning the given text into a predicate definition loc_axioms and deriving the original assumptions as local lemmas (modulo local definitions). The predicate statement covers only the newly specified assumptions, omitting the content of included locale expressions. The full cumulative view is only provided on export, involving another predicate loc that refers to the complete specification text.

In any case, the predicate arguments are those locale parameters that actually occur in the respective piece of text. Also note that these predicates operate at the meta-level in theory, but the locale packages attempts to internalize statements according to the object-logic setup (e.g. replacing \( \bigwedge \) by \( \forall \), and \( \rightarrow \rightarrow \) by \( \rightarrow \rightarrow \) in HOL; see also §9.5). Separate introduction rules loc_axioms.intro and loc.intro are provided as well.

print_locale locale prints the contents of the named locale. The command omits notes elements by default. Use print_locale! to have them included.

print_locales prints the names of all locales of the current theory.

intro_locales and unfold_locales repeatedly expand all introduction rules of locale predicates of the theory. While intro_locales only applies the loc.intro introduction rules and therefore does not descend to assumptions, unfold_locales is more aggressive and applies loc_axioms.intro as well. Both methods are aware of locale specifications entailed by the context, both from target statements, and from interpretations (see below). New goals that are entailed by the current context are discharged automatically.

5.6.3 Locale interpretations

<table>
<thead>
<tr>
<th>function</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>interpretation</td>
<td>theory ( \rightarrow ) proof(prove)</td>
</tr>
<tr>
<td>interpret</td>
<td>proof(state) | proof(chain) ( \rightarrow ) proof(prove)</td>
</tr>
<tr>
<td>sublocale</td>
<td>theory ( \rightarrow ) proof(prove)</td>
</tr>
<tr>
<td>print_dependencies*</td>
<td>context ( \rightarrow )</td>
</tr>
<tr>
<td>print_interps*</td>
<td>context ( \rightarrow )</td>
</tr>
</tbody>
</table>
Locale expressions may be instantiated, and the instantiated facts added to the current context. This requires a proof of the instantiated specification and is called \textit{locale interpretation}. Interpretation is possible in locales \texttt{sublocale}, theories (command \texttt{interpretation}) and also within a proof body (command \texttt{interpret}).
interpretation $expr$ where $eqns$ interprets $expr$ in the theory. The command generates proof obligations for the instantiated specifications (assumes and defines elements). Once these are discharged by the user, instantiated facts are added to the theory in a post-processing phase.

Additional equations, which are unfolded during post-processing, may be given after the keyword where. This is useful for interpreting concepts introduced through definitions. The equations must be proved.

The command is aware of interpretations already active in the theory, but does not simplify the goal automatically. In order to simplify the proof obligations use methods introlocales or unfoldlocales. Post-processing is not applied to facts of interpretations that are already active. This avoids duplication of interpreted facts, in particular. Note that, in the case of a locale with import, parts of the interpretation may already be active. The command will only process facts for new parts.

Adding facts to locales has the effect of adding interpreted facts to the theory for all interpretations as well. That is, interpretations dynamically participate in any facts added to locales. Note that if a theory inherits additional facts for a locale through one parent and an interpretation of that locale through another parent, the additional facts will not be interpreted.

interpret $expr$ where $eqns$ interprets $expr$ in the proof context and is otherwise similar to interpretation in theories. Note that rewrite rules given to interpret after the where keyword should be explicitly universally quantified.

sublocale $name \subseteq expr$ where $eqns$ interprets $expr$ in the locale $name$. A proof that the specification of $name$ implies the specification of $expr$ is required. As in the localized version of the theorem command, the proof is in the context of $name$. After the proof obligation has been discharged, the facts of $expr$ become part of locale $name$ as derived context elements and are available when the context $name$ is subsequently entered. Note that, like import, this is dynamic: facts added to a locale part of $expr$ after interpretation become also available in $name$.

Only specification fragments of $expr$ that are not already part of $name$ (be it imported, derived or a derived fragment of the import) are considered in this process. This enables circular interpretations provided that no infinite chains are generated in the locale hierarchy.
If interpretations of \textit{name} exist in the current theory, the command adds interpretations for \textit{expr} as well, with the same qualifier, although only for fragments of \textit{expr} that are not interpreted in the theory already.

Equations given after \textbf{where} amend the morphism through which \textit{expr} is interpreted. This enables to map definitions from the interpreted locales to entities of \textit{name}. This feature is experimental.

\texttt{print\_dependencies \textit{expr}} is useful for understanding the effect of an interpretation of \textit{expr}. It lists all locale instances for which interpretations would be added to the current context. Variant \texttt{print\_dependencies!} prints all locale instances that would be considered for interpretation, and would be interpreted in an empty context (that is, without interpretations).

\texttt{print\_interps \textit{locale}} lists all interpretations of \textit{locale} in the current theory or proof context, including those due to a combination of a \texttt{interpretation} or \texttt{interpret} and one or several \texttt{sublocale} declarations.

\begin{itemize}
  \item Since attributes are applied to interpreted theorems, interpretation may modify the context of common proof tools, e.g. the Simplifier or Classical Reasoner. As the behavior of such tools is \textbf{not} stable under interpretation morphisms, manual declarations might have to be added to the target context of the interpretation to revert such declarations.
  \item An interpretation in a theory or proof context may subsume previous interpretations. This happens if the same specification fragment is interpreted twice and the instantiation of the second interpretation is more general than the interpretation of the first. The locale package does not attempt to remove subsumed interpretations.
\end{itemize}

\section{5.7 Classes}

\begin{verbatim}
class : theory \to local\_theory
instantiation : theory \to local\_theory
instance : local\_theory \to local\_theory
instance : theory \to proof(prove)
subclass : local\_theory \to local\_theory
print\_classes*: context \to
class\_deps* : context \to
intro\_classes : method
\end{verbatim}
A class is a particular locale with \emph{exactly one} type variable $\alpha$. Beyond the underlying locale, a corresponding type class is established which is interpreted logically as axiomatic type class [44] whose logical content are the assumptions of the locale. Thus, classes provide the full generality of locales combined with the commodity of type classes (notably type-inference). See [14] for a short tutorial.
class $c = superclasses + body$ defines a new class $c$, inheriting from $superclasses$. This introduces a locale $c$ with import of all locales $superclasses$.

Any fixes in $body$ are lifted to the global theory level ($class operations f_1, \ldots, f_n$ of class $c$), mapping the local type parameter $\alpha$ to a schematic type variable $?\alpha :: c$.

Likewise, assumes in $body$ are also lifted, mapping each local parameter $f :: \tau[\alpha]$ to its corresponding global constant $f :: \tau[?\alpha :: c]$. The corresponding introduction rule is provided as $c.class_axioms.intro$. This rule should be rarely needed directly — the $intro_classes$ method takes care of the details of class membership proofs.

**instantiation** $t :: (s_1, \ldots, s_n)s$ begin opens a theory target (cf. §5.2) which allows to specify class operations $f_1, \ldots, f_n$ corresponding to sort $s$ at the particular type instance $(\alpha_1 :: s_1, \ldots, \alpha_n :: s_n) t$. A plain instance command in the target body poses a goal stating these type arities. The target is concluded by an end command.

Note that a list of simultaneous type constructors may be given; this corresponds nicely to mutually recursive type definitions, e.g. in Isabelle/HOL.

**instance** in an instantiation target body sets up a goal stating the type arities claimed at the opening instantiation. The proof would usually proceed by intro_classes, and then establish the characteristic theorems of the type classes involved. After finishing the proof, the background theory will be augmented by the proven type arities.

On the theory level, instance $t :: (s_1, \ldots, s_n)s$ provides a convenient way to instantiate a type class with no need to specify operations: one can continue with the instantiation proof immediately.

**subclass** $c$ in a class context for class $d$ sets up a goal stating that class $c$ is logically contained in class $d$. After finishing the proof, class $d$ is proven to be subclass $c$ and the locale $c$ is interpreted into $d$ simultaneously.

A weakened form of this is available through a further variant of instance: instance $c_1 \subseteq c_2$ opens a proof that class $c_2$ implies $c_1$. 
without reference to the underlying locales; this is useful if the properties to prove the logical connection are not sufficient on the locale level but on the theory level.

**print_classes** prints all classes in the current theory.

**class_deps** visualizes all classes and their subclass relations as a Hasse diagram.

**intro_classes** repeatedly expands all class introduction rules of this theory. Note that this method usually needs not be named explicitly, as it is already included in the default proof step (e.g. of **proof**). In particular, instantiation of trivial (syntactic) classes may be performed by a single “..” proof step.

### 5.7.1 The class target

A named context may refer to a locale (cf. §5.2). If this locale is also a class $c$, apart from the common locale target behaviour the following happens.

- Local constant declarations $g[\alpha]$ referring to the local type parameter $\alpha$ and local parameters $f[\alpha]$ are accompanied by theory-level constants $g[?\alpha :: c]$ referring to theory-level class operations $f[?\alpha :: c]$.
- Local theorem bindings are lifted as are assumptions.
- Local syntax refers to local operations $g[\alpha]$ and global operations $g[?\alpha :: c]$ uniformly. Type inference resolves ambiguities. In rare cases, manual type annotations are needed.

### 5.7.2 Co-regularity of type classes and arities

The class relation together with the collection of type-constructor arities must obey the principle of **co-regularity** as defined below.

For the subsequent formulation of co-regularity we assume that the class relation is closed by transitivity and reflexivity. Moreover the collection of arities $t :: (\exists) c$ is completed such that $t :: (\exists) c$ and $c \subseteq c'$ implies $t :: (\exists) c'$ for all such declarations.
Treating sorts as finite sets of classes (meaning the intersection), the class relation $c_1 \subseteq c_2$ is extended to sorts as follows:

$$s_1 \subseteq s_2 \equiv \forall c_2 \in s_2. \exists c_1 \in s_1. c_1 \subseteq c_2$$

This relation on sorts is further extended to tuples of sorts (of the same length) in the component-wise way.

Co-regularity of the class relation together with the arities relation means:

$$t :: (\overline{c}_1) \Rightarrow t :: (\overline{c}_2) \Rightarrow c_1 \subseteq c_2 \Rightarrow \overline{s}_1 \subseteq \overline{s}_2$$

for all such arities. In other words, whenever the result classes of some type-constructor arities are related, then the argument sorts need to be related in the same way.

Co-regularity is a very fundamental property of the order-sorted algebra of types. For example, it entails principle types and most general unifiers, e.g. see [27].

### 5.8 Unrestricted overloading

The overloading function

$$\text{overloading : theory} \rightarrow \text{local\_theory}$$

Isabelle/Pure’s definitional schemes support certain forms of overloading (see §5.10.3). Overloading means that a constant being declared as $c :: \alpha \ \text{decl}$ may be defined separately on type instances $c :: (\beta_1, \ldots, \beta_n) \ t \ \text{decl}$ for each type constructor $t$. At most occasions overloading will be used in a Haskell-like fashion together with type classes by means of **instantiation** (see §5.7). Sometimes low-level overloading is desirable. The overloading target provides a convenient view for end-users.

```
overloading spec begin

spec

name \equiv term

≡ unchecked
```
overloading \( x_1 \equiv c_1 :: \tau_1 \) and \( \ldots x_n \equiv c_n :: \tau_n \) begin opens a theory target (cf. §5.2) which allows to specify constants with overloaded definitions. These are identified by an explicitly given mapping from variable names \( x_i \) to constants \( c_i \) at particular type instances. The definitions themselves are established using common specification tools, using the names \( x_i \) as reference to the corresponding constants. The target is concluded by \textbf{end}.

A (\textit{unchecked}) option disables global dependency checks for the corresponding definition, which is occasionally useful for exotic overloading (see §5.10.3 for a precise description). It is at the discretion of the user to avoid malformed theory specifications!

### 5.9 Incorporating ML code

\begin{verbatim}
use : local_theory \rightarrow \ local_theory
ML : local_theory \rightarrow \ local_theory
ML_prf : proof \rightarrow \ proof
ML_val : any \rightarrow
ML_command : any \rightarrow
setup : theory \rightarrow \ theory
local_setup : local_theory \rightarrow \ local_theory
attribute_setup : theory \rightarrow \ theory
\end{verbatim}
**use file** reads and executes ML commands from file. The current theory context is passed down to the ML toplevel and may be modified, using `Context.>>` or derived ML commands. The file name is checked with the `uses` dependency declaration given in the theory header (see also §5.1).

Top-level ML bindings are stored within the (global or local) theory context.

**ML text** is similar to `use`, but executes ML commands directly from the given `text`. Top-level ML bindings are stored within the (global or local) theory context.

**ML_prf** is analogous to **ML** but works within a proof context.

Top-level ML bindings are stored within the proof context in a purely sequential fashion, disregarding the nested proof structure. ML bindings introduced by **ML_prf** are discarded at the end of the proof.

**ML_val** and **ML_command** are diagnostic versions of **ML**, which means that the context may not be updated. **ML_val** echos the bindings produced at the ML toplevel, but **ML_command** is silent.

**setup text** changes the current theory context by applying `text`, which refers to an ML expression of type `theory -> theory`. This enables to initialize any object-logic specific tools and packages written in ML, for example.

**local_setup** is similar to **setup** for a local theory context, and an ML expression of type `local_theory -> local_theory`. This allows to invoke local theory specification packages without going through concrete outer syntax, for example.

**attribute_setup** `name = text description` defines an attribute in the current theory. The given `text` has to be an ML expression of type `attribute context_parser`, cf. basic parsers defined in structure `Args` and `Attrib`.

In principle, attributes can operate both on a given theorem and the implicit context, although in practice only one is modified and the other serves as parameter. Here are examples for these two cases:
attribute_setup my_rule = {*
  Attrib.thms >> (fn ths =>
    Thm.rule_attribute
    (fn context: Context.generic => fn th: thm =>
      let val th’ = th OF ths
      in th’ end)) *}

attribute_setup my_declaration = {*
  Attrib.thms >> (fn ths =>
    Thm.declaration_attribute
    (fn th: thm => fn context: Context.generic =>
      let val context’ = context
      in context’ end)) *}

5.10 Primitive specification elements

5.10.1 Type classes and sorts

- **classes** : theory → theory
- **classrel** : theory → theory (axiomatic!)
- **default_sort** : local_theory → local_theory

classes c ⊆ c₁, ... cₙ declares class c to be a subclass of existing classes c₁, ..., cₙ. Isabelle implicitly maintains the transitive closure of the class hierarchy. Cyclic class structures are not permitted.
classrel $c_1 \subseteq c_2$ states subclass relations between existing classes $c_1$ and $c_2$. This is done axiomatically! The subclass and instance commands (see §5.7) provide a way to introduce proven class relations.

default_sort $s$ makes sort $s$ the new default sort for any type variable that is given explicitly in the text, but lacks a sort constraint (wrt. the current context). Type variables generated by type inference are not affected.

Usually the default sort is only changed when defining a new object-logic. For example, the default sort in Isabelle/HOL is $type$, the class of all HOL types.

When merging theories, the default sorts of the parents are logically intersected, i.e. the representations as lists of classes are joined.

### 5.10.2 Types and type abbreviations

type_synonym : $local\_theory \rightarrow local\_theory$

typedecl : $local\_theory \rightarrow local\_theory$

arities : $theory \rightarrow theory$ (axiomatic!)

type_synonym $(\alpha_1, \ldots, \alpha_n) t = \tau$ introduces a type synonym $(\alpha_1, \ldots, \alpha_n) t$ for the existing type $\tau$. Unlike actual type definitions, as are available in Isabelle/HOL for example, type synonyms are merely syntactic abbreviations without any logical significance. Internally, type synonyms are fully expanded.
typedcl $(\alpha_1, \ldots, \alpha_n)$ $t$ declares a new type constructor $t$. If the object-logic defines a base sort $s$, then the constructor is declared to operate on that, via the axiomatic specification arities $t :: (s, \ldots, s)s$.

arities $t :: (s_1, \ldots, s_n)s$ augments Isabelle’s order-sorted signature of types by new type constructor arities. This is done axiomatically! The instantiation target (see §5.7) provides a way to introduce proven type arities.

### 5.10.3 Constants and definitions

- **consts** : $\text{theory} \rightarrow \text{theory}$
- **defs** : $\text{theory} \rightarrow \text{theory}$

Definitions essentially express abbreviations within the logic. The simplest form of a definition is $c :: \sigma \equiv t$, where $c$ is a newly declared constant. Isabelle also allows derived forms where the arguments of $c$ appear on the left, abbreviating a prefix of $\lambda$-abstractions, e.g. $c \equiv \lambda x \ y. \ t$ may be written more conveniently as $c \ x \ y \equiv t$. Moreover, definitions may be weakened by adding arbitrary pre-conditions: $A \Rightarrow c \ x \ y \equiv t$.

The built-in well-formedness conditions for definitional specifications are:

- Arguments (on the left-hand side) must be distinct variables.
- All variables on the right-hand side must also appear on the left-hand side.
- All type variables on the right-hand side must also appear on the left-hand side; this prohibits $0 :: \text{nat} \equiv \text{length} ([] :: \alpha \text{list})$ for example.
- The definition must not be recursive. Most object-logics provide definitional principles that can be used to express recursion safely.

The right-hand side of overloaded definitions may mention overloaded constants recursively at type instances corresponding to the immediate argument types $\beta_1, \ldots, \beta_n$. Incomplete specification patterns impose global constraints on all occurrences, e.g. $d :: \alpha \times \alpha$ on the left-hand side means that all corresponding occurrences on some right-hand side need to be an instance of this, general $d :: \alpha \times \beta$ will be disallowed.
5.11 Axioms and theorems

axioms : theory → theory (axiomatic!)
lemmas : local_theory → local_theory
theorems : local_theory → local_theory
axioms $\varphi$ introduces arbitrary statements as axioms of the meta-logic. In fact, axioms are “axiomatic theorems”, and may be referred later just as any other theorem. Axioms are usually only introduced when declaring new logical systems. Everyday work is typically done the hard way, with proper definitions and proven theorems.

lemmas $a = b_1 \ldots b_n$ for $x_1 \ldots x_m$ evaluates given facts (with attributes) in the current context, which may be augmented by local variables. Results are standardized before being stored, i.e. schematic variables are renamed to enforce index 0 uniformly.

theorems is the same as lemmas, but marks the result as a different kind of facts.

5.12 Oracles

oracle : theory $\rightarrow$ theory (axiomatic!)

Oracles allow Isabelle to take advantage of external reasoners such as arithmetic decision procedures, model checkers, fast tautology checkers or computer algebra systems. Invoked as an oracle, an external reasoner can create arbitrary Isabelle theorems.
It is the responsibility of the user to ensure that the external reasoner is as trustworthy as the application requires. Another typical source of errors is the linkup between Isabelle and the external tool, not just its concrete implementation, but also the required translation between two different logical environments.

Isabelle merely guarantees well-formedness of the propositions being asserted, and records within the internal derivation object how presumed theorems depend on unproven suppositions.

\[
\text{oracle name} = \text{text}
\]

\text{oracle name} = \text{text} turns the given ML expression \text{text} of type \('a \rightarrow \text{cterm}' into an ML function of type \('a \rightarrow \text{thm}'\), which is bound to the global identifier \text{name}. This acts like an infinitary specification of axioms! Invoking the oracle only works within the scope of the resulting theory.

See ~/src/HOL/ex/Iff_Oracle.thy for a worked example of defining a new primitive rule as oracle, and turning it into a proof method.

### 5.13 Name spaces

- \text{hide_class} : \text{theory} \rightarrow \text{theory}
- \text{hide_type} : \text{theory} \rightarrow \text{theory}
- \text{hide_const} : \text{theory} \rightarrow \text{theory}
- \text{hide_fact} : \text{theory} \rightarrow \text{theory}

Isabelle organizes any kind of name declarations (of types, constants, theorems etc.) by separate hierarchically structured name spaces. Normally the user does not have to control the behavior of name spaces by hand, yet the following commands provide some way to do so.
hide_class names fully removes class declarations from a given name space; with the (open) option, only the base name is hidden.

Note that hiding name space accesses has no impact on logical declarations — they remain valid internally. Entities that are no longer accessible to the user are printed with the special qualifier “??” prefixed to the full internal name.

hide_type, hide_const, and hide_fact are similar to hide_class, but hide types, constants, and facts, respectively.
Proof commands perform transitions of Isar/VM machine configurations, which are block-structured, consisting of a stack of nodes with three main components: logical proof context, current facts, and open goals. Isar/VM transitions are typed according to the following three different modes of operation:

- **proof**(prove) means that a new goal has just been stated that is now to be proven; the next command may refine it by some proof method, and enter a sub-proof to establish the actual result.

- **proof**(state) is like a nested theory mode: the context may be augmented by stating additional assumptions, intermediate results etc.

- **proof**(chain) is intermediate between **proof**(state) and **proof**(prove): existing facts (i.e. the contents of the special “this” register) have been just picked up in order to be used when refining the goal claimed next.

The proof mode indicator may be understood as an instruction to the writer, telling what kind of operation may be performed next. The corresponding typings of proof commands restricts the shape of well-formed proof texts to particular command sequences. So dynamic arrangements of commands eventually turn out as static texts of a certain structure.

Appendix A gives a simplified grammar of the (extensible) language emerging that way from the different types of proof commands. The main ideas of the overall Isar framework are explained in chapter 2.

### 6.1 Proof structure

#### 6.1.1 Formal notepad

\[
\text{notepad} : \text{local\_theory} \rightarrow \text{proof\(\text{(state)}\)}
\]
notepad begin

end

notepad begin opens a proof state without any goal statement. This allows to experiment with Isar, without producing any persistent result.

The notepad can be closed by end or discontinued by oops.

6.1.2 Blocks

\textbf{next}: proof(state) → proof(state)
\{ : proof(state) → proof(state)
\} : proof(state) → proof(state)

While Isar is inherently block-structured, opening and closing blocks is mostly handled rather casually, with little explicit user-intervention. Any local goal statement automatically opens two internal blocks, which are closed again when concluding the sub-proof (by qed etc.). Sections of different context within a sub-proof may be switched via next, which is just a single block-close followed by block-open again. The effect of next is to reset the local proof context; there is no goal focus involved here!

For slightly more advanced applications, there are explicit block parentheses as well. These typically achieve a stronger forward style of reasoning.

next switches to a fresh block within a sub-proof, resetting the local context to the initial one.

\{ and \} explicitly open and close blocks. Any current facts pass through “{” unchanged, while “}” causes any result to be exported into the enclosing context. Thus fixed variables are generalized, assumptions discharged, and local definitions unfolded (cf. §6.2.1). There is no difference of assume and presume in this mode of forward reasoning — in contrast to plain backward reasoning with the result exported at show time.
6.1.3 Omitting proofs

\texttt{oops} : \textit{proof} $\rightarrow$ \textit{local\_theory} $|$ \textit{theory}

The \texttt{oops} command discontinues the current proof attempt, while considering the partial proof text as properly processed. This is conceptually quite different from “faking” actual proofs via \texttt{sorry} (see §6.3.2): \texttt{oops} does not observe the proof structure at all, but goes back right to the theory level. Furthermore, \texttt{oops} does not produce any result theorem — there is no intended claim to be able to complete the proof in any way.

A typical application of \texttt{oops} is to explain Isar proofs within the system itself, in conjunction with the document preparation tools of Isabelle described in chapter 4. Thus partial or even wrong proof attempts can be discussed in a logically sound manner. Note that the Isabelle \TeX macros can be easily adapted to print something like “…” instead of the keyword “\texttt{oops}”.

6.2 Statements

6.2.1 Context elements

\begin{verbatim}
fix    : proof(state) $\rightarrow$ proof(state)
assume : proof(state) $\rightarrow$ proof(state)
presume: proof(state) $\rightarrow$ proof(state)
def    : proof(state) $\rightarrow$ proof(state)
\end{verbatim}

The logical proof context consists of fixed variables and assumptions. The former closely correspond to Skolem constants, or meta-level universal quantification as provided by the Isabelle/Pure logical framework. Introducing some arbitrary, but fixed variable via “fix $x$” results in a local value that may be used in the subsequent proof as any other variable or constant. Furthermore, any result $\vdash \varphi[x]$ exported from the context will be universally closed wrt. $x$ at the outermost level: $\vdash \forall x. \varphi[x]$ (this is expressed in normal form using Isabelle’s meta-variables).

Similarly, introducing some assumption $\chi$ has two effects. On the one hand, a local theorem is created that may be used as a fact in subsequent proof steps. On the other hand, any result $\chi \vdash \varphi$ exported from the context becomes conditional wrt. the assumption: $\vdash \chi \Rightarrow \varphi$. Thus, solving an enclosing goal using such a result would basically introduce a new subgoal stemming from the assumption. How this situation is handled depends on the version of assumption command used: while \texttt{assume} insists on solving the subgoal
by unification with some premise of the goal, \texttt{presume} leaves the subgoal unchanged in order to be proved later by the user.

Local definitions, introduced by “\texttt{def }x \equiv t\texttt{ }”, are achieved by combining “\texttt{fix }x\texttt{ }” with another version of assumption that causes any hypothetical equation \(x \equiv t\) to be eliminated by the reflexivity rule. Thus, exporting some result \(x \equiv t \vdash \varphi[x]\) yields \(\vdash \varphi[t]\).

\[\text{fix }\]

\[\text{vars}\]

\[\text{and}\]

\[\text{assume}\]

\[\text{props}\]

\[\text{and}\]

\[\text{presume}\]

\[\text{and}\]

\[\text{def}\]

\[\text{def}\]

\[\text{and}\]

\[\text{def}\]

\[\text{thmdecl}\]

\[\text{name} \equiv \text{term}\]

\[\equiv\]

\[\text{term}_\text{pat}\]

\text{fix } x \text{ introduces a local variable } x \text{ that is arbitrary, but fixed.}

\textbf{assume } a: \varphi \text{ and } \textbf{presume } a: \varphi \text{ introduce a local fact } \varphi \vdash \varphi \text{ by assumption. Subsequent results applied to an enclosing goal (e.g. by show) are handled as follows: assume expects to be able to unify with existing premises in the goal, while presume leaves } \varphi \text{ as new subgoals. Several lists of assumptions may be given (separated by and; the resulting list of current facts consists of all of these concatenated.}
**CHAPTER 6. PROOFS**

**def** \( x \equiv t \) introduces a local (non-polymorphic) definition. In results exported from the context, \( x \) is replaced by \( t \). Basically, “**def** \( x \equiv t \)” abbreviates “**fix** \( x \) **assume** \( x \equiv t \)”, with the resulting hypothetical equation solved by reflexivity.

The default name for the definitional equation is \( x.def \). Several simultaneous definitions may be given at the same time.

The special name *prems* refers to all assumptions of the current context as a list of theorems. This feature should be used with great care! It is better avoided in final proof texts.

### 6.2.2 Term abbreviations

**let** : \( proof(state) \rightarrow proof(state) \)

**is** : \( syntax \)

Abbreviations may be either bound by explicit **let** \( p \equiv t \) statements, or by annotating assumptions or goal statements with a list of patterns “(**is** \( p_1 \ldots p_n \))”. In both cases, higher-order matching is invoked to bind extra-logical term variables, which may be either named schematic variables of the form \( ?x \), or nameless dummies “_” (underscore). Note that in the **let** form the patterns occur on the left-hand side, while the **is** patterns are in postfix position.

Polymorphism of term bindings is handled in Hindley-Milner style, similar to ML. Type variables referring to local assumptions or open goal statements are **fixed**, while those of finished results or bound by **let** may occur in arbitrary instances later. Even though actual polymorphism should be rarely used in practice, this mechanism is essential to achieve proper incremental type-inference, as the user proceeds to build up the Isar proof text from left to right.

Term abbreviations are quite different from local definitions as introduced via **def** (see §6.2.1). The latter are visible within the logic as actual equations, while abbreviations disappear during the input process just after type checking. Also note that **def** does not support polymorphism.
The syntax of is patterns follows termPat or propPat (see §3.2.6).

\[ \text{let } p_1 = t_1 \text{ and } \ldots p_n = t_n \text{ binds any text variables in patterns } p_1, \ldots, p_n \text{ by simultaneous higher-order matching against terms } t_1, \ldots, t_n. \]

(is \( p_1 \ldots p_n \)) resembles let, but matches \( p_1, \ldots, p_n \) against the preceding statement. Also note that is is not a separate command, but part of others (such as assume, have etc.).

Some implicit term abbreviations for goals and facts are available as well. For any open goal, thesis refers to its object-level statement, abstracted over any meta-level parameters (if present). Likewise, this is bound for fact statements resulting from assumptions or finished goals. In case this refers to an object-logic statement that is an application \( f \ t \), then \( t \) is bound to the special text variable “…” (three dots). The canonical application of this convenience are calculational proofs (see §6.5).

### 6.2.3 Facts and forward chaining

- **Note**: \( \text{proof(state)} \rightarrow \text{proof(state)} \)
- **Then**: \( \text{proof(state)} \rightarrow \text{proof(chain)} \)
- **From**: \( \text{proof(state)} \rightarrow \text{proof(chain)} \)
- **With**: \( \text{proof(state)} \rightarrow \text{proof(chain)} \)
- **Using**: \( \text{proof(prove)} \rightarrow \text{proof(prove)} \)
- **Unfolding**: \( \text{proof(prove)} \rightarrow \text{proof(prove)} \)

New facts are established either by assumption or proof of local statements. Any fact will usually be involved in further proofs, either as explicit arguments of proof methods, or when forward chaining towards the next goal via then (and variants); from and with are composite forms involving note. The using elements augments the collection of used facts after a goal has been stated. Note that the special theorem name this refers to the most recently established facts, but only before issuing a follow-up claim.
CHAPTER 6. PROOFS

**note** $a = b_1 \ldots b_n$ recalls existing facts $b_1, \ldots, b_n$, binding the result as $a$. Note that attributes may be involved as well, both on the left and right hand sides.

**then** indicates forward chaining by the current facts in order to establish the goal to be claimed next. The initial proof method invoked to refine that will be offered the facts to do “anything appropriate” (see also §6.3.2). For example, method *rule* (see §6.3.3) would typically do an elimination rather than an introduction. Automatic methods usually insert the facts into the goal state before operation. This provides a simple scheme to control relevance of facts in automated proof search.

**from** $b$ abbreviates “**note** $b$ **then**”; thus **then** is equivalent to “**from** this”.

**with** $b_1 \ldots b_n$ abbreviates “**from** $b_1 \ldots b_n$ and this”; thus the forward chaining is from earlier facts together with the current ones.

**using** $b_1 \ldots b_n$ augments the facts being currently indicated for use by a subsequent refinement step (such as *apply* or *proof*).

**unfolding** $b_1 \ldots b_n$ is structurally similar to **using**, but unfolds definitional equations $b_1, \ldots, b_n$ throughout the goal state and facts.

Forward chaining with an empty list of theorems is the same as not chaining at all. Thus “**from** nothing” has no effect apart from entering *prove(chain)* mode, since *nothing* is bound to the empty list of theorems.

Basic proof methods (such as *rule*) expect multiple facts to be given in their proper order, corresponding to a prefix of the premises of the rule involved. Note that positions may be easily skipped using something like **from _ and a and b**, for example. This involves the trivial rule $PROP \psi \Rightarrow PROP \psi$, which is bound in Isabelle/Pure as “_” (underscore).

Automated methods (such as *simp* or *auto*) just insert any given facts before their usual operation. Depending on the kind of procedure involved, the order of facts is less significant here.
6.2.4 Goals

lemma : $\text{local\_theory} \rightarrow \text{proof}(\text{prove})$

theorem : $\text{local\_theory} \rightarrow \text{proof}(\text{prove})$

corollary : $\text{local\_theory} \rightarrow \text{proof}(\text{prove})$

schematic_lemma : $\text{local\_theory} \rightarrow \text{proof}(\text{prove})$

schematic_theorem : $\text{local\_theory} \rightarrow \text{proof}(\text{prove})$

schematic_corollary : $\text{local\_theory} \rightarrow \text{proof}(\text{prove})$

have : $\text{proof}(\text{state}) \mid \text{proof}(\text{chain}) \rightarrow \text{proof}(\text{prove})$

show : $\text{proof}(\text{state}) \mid \text{proof}(\text{chain}) \rightarrow \text{proof}(\text{prove})$

hence : $\text{proof}(\text{state}) \rightarrow \text{proof}(\text{prove})$

thus : $\text{proof}(\text{state}) \rightarrow \text{proof}(\text{prove})$

print_statement* : $\text{context} \rightarrow$

From a theory context, proof mode is entered by an initial goal command such as lemma, theorem, or corollary. Within a proof, new claims may be introduced locally as well; four variants are available here to indicate whether forward chaining of facts should be performed initially (via then), and whether the final result is meant to solve some pending goal.

Goals may consist of multiple statements, resulting in a list of facts eventually. A pending multi-goal is internally represented as a meta-level conjunction ($\&\&$), which is usually split into the corresponding number of sub-goals prior to an initial method application, via proof (§6.3.2) or apply (§6.3.4). The induct method covered in §6.6 acts on multiple claims simultaneously.

Claims at the theory level may be either in short or long form. A short goal merely consists of several simultaneous propositions (often just one). A long goal includes an explicit context specification for the subsequent conclusion, involving local parameters and assumptions. Here the role of each part of the statement is explicitly marked by separate keywords (see also §5.6); the local assumptions being introduced here are available as assms in the proof. Moreover, there are two kinds of conclusions: shows states several simultaneous propositions (essentially a big conjunction), while obtains claims several simultaneous simultaneous contexts of (essentially a big disjunction of eliminated parameters and assumptions, cf. §6.4).
**Conclusion**

- **shows** goal
- **obtains** case
- **parname**
- **case**

**Case**

- **vars**
- **where**
- **props**

**Lemma** $a$: $\varphi$ enters proof mode with $\varphi$ as main goal, eventually resulting in some fact $\vdash \varphi$ to be put back into the target context. An additional context specification may build up an initial proof context for the subsequent claim; this includes local definitions and syntax as well, see also includes in §5.3 and context_elem in §5.6.

**Theorem** $a$: $\varphi$ and **Corollary** $a$: $\varphi$ are essentially the same as **Lemma** $a$: $\varphi$, but the facts are internally marked as being of a different kind. This discrimination acts like a formal comment.

**Schematic Lemma**, **Schematic Theorem**, **Schematic Corollary** are similar to **Lemma**, **Theorem**, **Corollary**, respectively but allow the statement to contain unbound schematic variables.

Under normal circumstances, an Isar proof text needs to specify claims explicitly. Schematic goals are more like goals in Prolog, where certain results are synthesized in the course of reasoning. With schematic statements, the inherent compositionality of Isar proofs is lost, which also impacts performance, because proof checking is forced into sequential mode.

**Have** $a$: $\varphi$ claims a local goal, eventually resulting in a fact within the current logical context. This operation is completely independent of any pending sub-goals of an enclosing goal statements, so **Have** may be freely used for experimental exploration of potential results within a proof body.
show $a$: $\varphi$ is like have $a$: $\varphi$ plus a second stage to refine some pending sub-goal for each one of the finished result, after having been exported into the corresponding context (at the head of the sub-proof of this show command).

To accommodate interactive debugging, resulting rules are printed before being applied internally. Even more, interactive execution of show predicts potential failure and displays the resulting error as a warning beforehand. Watch out for the following message:

Problem! Local statement will fail to solve any pending goal

hence abbreviates “then have”, i.e. claims a local goal to be proven by forward chaining the current facts. Note that hence is also equivalent to “from this have”.

thus abbreviates “then show”. Note that thus is also equivalent to “from this show”.

print_statement $a$ prints facts from the current theory or proof context in long statement form, according to the syntax for lemma given above.

Any goal statement causes some term abbreviations (such as $\textit{thesis}$) to be bound automatically, see also §6.2.2.

The optional case names of obtains have a twofold meaning: (1) during the of this claim they refer to the the local context introductions, (2) the resulting rule is annotated accordingly to support symbolic case splits when used with the cases method (cf. §6.6).

6.3 Refinement steps

6.3.1 Proof method expressions

Proof methods are either basic ones, or expressions composed of methods via “,” (sequential composition), “|” (alternative choices), “?” (try), “+” (repeat at least once), “[n]” (restriction to first $n$ sub-goals, with default $n = 1$). In practice, proof methods are usually just a comma separated list of nameref args specifications. Note that parentheses may be dropped for single method specifications (with no arguments).
Proper Isar proof methods do not admit arbitrary goal addressing, but refer either to the first sub-goal or all sub-goals uniformly. The goal restriction operator “[n]” evaluates a method expression within a sandbox consisting of the first n sub-goals (which need to exist). For example, the method “simp_all[3]” simplifies the first three sub-goals, while “(rule foo, simp_all)[]” simplifies all new goals that emerge from applying rule foo to the originally first one.

Improper methods, notably tactic emulations, offer a separate low-level goal addressing scheme as explicit argument to the individual tactic being involved. Here “[!]” refers to all goals, and “[n−]” to all goals starting from n.
6.3.2 Initial and terminal proof steps

- **proof**: \( \text{proof}(\text{prove}) \to \text{proof}(\text{state}) \)
- **qed**: \( \text{proof}(\text{state}) \to \text{proof}(\text{state}) \mid \text{local\_theory} \mid \text{theory} \)
- **by**: \( \text{proof}(\text{prove}) \to \text{proof}(\text{state}) \mid \text{local\_theory} \mid \text{theory} \)
- **..**: \( \text{proof}(\text{prove}) \to \text{proof}(\text{state}) \mid \text{local\_theory} \mid \text{theory} \)
- **.:**: \( \text{proof}(\text{prove}) \to \text{proof}(\text{state}) \mid \text{local\_theory} \mid \text{theory} \)
- **sorry**: \( \text{proof}(\text{prove}) \to \text{proof}(\text{state}) \mid \text{local\_theory} \mid \text{theory} \)

Arbitrary goal refinement via tactics is considered harmful. Structured proof composition in Isar admits proof methods to be invoked in two places only.

1. An initial refinement step **proof** \( m_1 \) reduces a newly stated goal to a number of sub-goals that are to be solved later. Facts are passed to \( m_1 \) for forward chaining, if so indicated by **proof(chain)** mode.

2. A terminal conclusion step **qed** \( m_2 \) is intended to solve remaining goals. No facts are passed to \( m_2 \).

The only other (proper) way to affect pending goals in a proof body is by **show**, which involves an explicit statement of what is to be solved eventually. Thus we avoid the fundamental problem of unstructured tactic scripts that consist of numerous consecutive goal transformations, with invisible effects.

As a general rule of thumb for good proof style, initial proof methods should either solve the goal completely, or constitute some well-understood reduction to new sub-goals. Arbitrary automatic proof tools that are prone leave a large number of badly structured sub-goals are no help in continuing the proof document in an intelligible manner.
Unless given explicitly by the user, the default initial method is rule (or its classical variant rule), which applies a single standard elimination or introduction rule according to the topmost symbol involved. There is no separate default terminal method. Any remaining goals are always solved by assumption in the very last step.

**proof** $m_1$ refines the goal by proof method $m_1$; facts for forward chaining are passed if so indicated by proof(chain) mode.

**qed** $m_2$ refines any remaining goals by proof method $m_2$ and concludes the sub-proof by assumption. If the goal had been show (or thus), some pending sub-goal is solved as well by the rule resulting from the result exported into the enclosing goal context. Thus qed may fail for two reasons: either $m_2$ fails, or the resulting rule does not fit to any pending goal\(^1\) of the enclosing context. Debugging such a situation might involve temporarily changing show into have, or weakening the local context by replacing occurrences of assume by presume.

---

\(^1\)This includes any additional “strong” assumptions as introduced by assume.
by $m_1 \ m_2$ is a terminal proof; it abbreviates proof $m_1 \ \text{qed} \ m_2$, but with backtracking across both methods. Debugging an unsuccessful by $m_1 \ m_2$ command can be done by expanding its definition; in many cases proof $m_1$ (or even apply $m_1$) is already sufficient to see the problem.

“..” is a default proof; it abbreviates by rule.

“.” is a trivial proof; it abbreviates by this.

sorry is a fake proof pretending to solve the pending claim without further ado. This only works in interactive development, or if the quick_and_dirty flag is enabled (in ML). Facts emerging from fake proofs are not the real thing. Internally, each theorem container is tainted by an oracle invocation, which is indicated as “[!]” in the printed result.

The most important application of sorry is to support experimentation and top-down proof development.

6.3.3 Fundamental methods and attributes

The following proof methods and attributes refer to basic logical operations of Isar. Further methods and attributes are provided by several generic and object-logic specific tools and packages (see chapter 9 and chapter 10).

- : method
fact : method
assumption : method
this : method
rule : method
intro : attribute
elim : attribute
dest : attribute
rule : attribute
OF : attribute
of : attribute
where : attribute
“−” (minus) does nothing but insert the forward chaining facts as premises into the goal. Note that command \texttt{proof} without any method actually performs a single reduction step using the \textit{rule} method; thus a plain \textit{do-nothing} proof step would be \texttt{proof −} rather than \texttt{proof} alone.

\textit{fact} $a_1 \ldots a_n$ composes some fact from $a_1, \ldots, a_n$ (or implicitly from the current proof context) modulo unification of schematic type and term variables. The rule structure is not taken into account, i.e. meta-level implication is considered atomic. This is the same principle underlying literal facts (cf. §3.2.7): \texttt{have $\varphi$ by fact} is equivalent to \texttt{note ‘$\varphi$’} provided that \( \vdash \varphi \) is an instance of some known \( \vdash \varphi \) in the proof context.

\textit{assumption} solves some goal by a single assumption step. All given facts are guaranteed to participate in the refinement; this means there may be only 0 or 1 in the first place. Recall that \texttt{qed} (§6.3.2) already concludes any remaining sub-goals by assumption, so structured proofs usually need not quote the \textit{assumption} method at all.

\textit{this} applies all of the current facts directly as rules. Recall that "." (dot) abbreviates \textit{by this}.

\textit{rule} $a_1 \ldots a_n$ applies some rule given as argument in backward manner; facts are used to reduce the rule before applying it to the goal. Thus \textit{rule} without facts is plain introduction, while with facts it becomes elimination.

When no arguments are given, the \textit{rule} method tries to pick appropriate rules automatically, as declared in the current context using the \textit{intro}, \textit{elim}, \textit{dest} attributes (see below). This is the default behavior of \texttt{proof} and ".." (double-dot) steps (see §6.3.2).
intro, elim, and dest declare introduction, elimination, and destruct rules, to be used with method rule, and similar tools. Note that the latter will ignore rules declared with “?” , while “!” are used most aggressively.

The classical reasoner (see §9.4) introduces its own variants of these attributes; use qualified names to access the present versions of Isabelle/Pure, i.e. Pure.intro.

rule del undeclares introduction, elimination, or destruct rules.

OF a₁ ... aₙ applies some theorem to all of the given rules a₁, ..., aₙ in canonical right-to-left order, which means that premises stemming from the aᵢ emerge in parallel in the result, without interfering with each other. In many practical situations, the aᵢ do not have premises themselves, so rule [OF a₁ ... aₙ] can be actually read as functional application (modulo unification).

Argument positions may be effectively skipped by using “_” (underscore), which refers to the propositional identity rule in the Pure theory.

of t₁ ... tₙ performs positional instantiation of term variables. The terms t₁, ..., tₙ are substituted for any schematic variables occurring in a theorem from left to right; “_” (underscore) indicates to skip a position. Arguments following a “concl:” specification refer to positions of the conclusion of a rule.

where x₁ = t₁ and ... xₙ = tₙ performs named instantiation of schematic type and term variables occurring in a theorem. Schematic variables have to be specified on the left-hand side (e.g. ?x1.3). The question mark may be omitted if the variable name is a plain identifier without index. As type instantiations are inferred from term instantiations, explicit type instantiations are seldom necessary.

6.3.4 Emulating tactic scripts

The Isar provides separate commands to accommodate tactic-style proof scripts within the same system. While being outside the orthodox Isar proof language, these might come in handy for interactive exploration and debugging, or even actual tactical proof within new-style theories (to benefit from document preparation, for example). See also §9.2.3 for actual tactics, that
have been encapsulated as proof methods. Proper proof methods may be
used in scripts, too.

apply*: \( \text{proof}\text{prove} \rightarrow \text{proof}\text{prove} \)
apply_end*: \( \text{proof}\text{state} \rightarrow \text{proof}\text{state} \)
done*: \( \text{proof}\text{prove} \rightarrow \text{proof}\text{state} \mid \text{local}\text{theory} \mid \text{theory} \)
defer*: \( \text{proof} \rightarrow \text{proof} \)
prefer*: \( \text{proof} \rightarrow \text{proof} \)
back*: \( \text{proof} \rightarrow \text{proof} \)

apply \( m \) applies proof method \( m \) in initial position, but unlike \( \text{proof} \) it
retains \( \text{proof}\text{prove} \) mode. Thus consecutive method applications
may be given just as in tactic scripts.
Facts are passed to \( m \) as indicated by the goal’s forward-chain mode,
and are consumed afterwards. Thus any further apply command would
always work in a purely backward manner.

apply_end \( m \) applies proof method \( m \) as if in terminal position. Basically,
this simulates a multi-step tactic script for \text{qed}, but may be given
anywhere within the proof body.
No facts are passed to \( m \) here. Furthermore, the static context is that of
the enclosing goal (as for actual \text{qed}). Thus the proof method may not
refer to any assumptions introduced in the current body, for example.

done completes a proof script, provided that the current goal state is solved
completely. Note that actual structured proof commands (e.g. \text{.} or
\text{sorry}) may be used to conclude proof scripts as well.
defer \( n \) and prefer \( n \) shuffle the list of pending goals: defer puts off sub-goal \( n \) to the end of the list (\( n = 1 \) by default), while prefer brings sub-goal \( n \) to the front.

back does back-tracking over the result sequence of the latest proof command. Basically, any proof command may return multiple results.

Any proper Isar proof method may be used with tactic script commands such as apply. A few additional emulations of actual tactics are provided as well; these would be never used in actual structured proofs, of course.

### 6.3.5 Defining proof methods

\[
\text{method_setup} : \text{theory} \rightarrow \text{theory}
\]

\[
\text{method_setup} \quad \text{name} = \text{text description}
\]

method_setup \( name = text \) description defines a proof method in the current theory. The given text has to be an ML expression of type \((\text{Proof.context} \rightarrow \text{Proof.method})\) context_parser, cf. basic parsers defined in structure \text{Args} and \text{Attrib}. There are also combinators like METHOD and SIMPLE_METHOD to turn certain tactic forms into official proof methods; the primed versions refer to tactics with explicit goal addressing.

Here are some example method definitions:

method_setup \( my\_method1 = \{\}
    \quad \text{Scan.succeed (K \ (\text{SIMPLE\_METHOD}' (fn \ i: \text{int} => \text{no\_tac})))}
\}
\) "my first method (without any arguments)"

method_setup \( my\_method2 = \{\}
    \quad \text{Scan.succeed (fn \ ctxt: \text{Proof.context} =>}
    \quad \quad \text{SIMPLE\_METHOD' (fn \ i: \text{int} => \text{no\_tac}))}
\}
\) "my second method (with context)"

method_setup \( my\_method3 = \{\}
    \quad \text{Attrib.thms >> (fn \ thms: \text{thm list} => fn \ ctxt: \text{Proof.context} =>}
    \quad \quad \text{SIMPLE\_METHOD' (fn \ i: \text{int} => \text{no\_tac}))}
\}
\) "my third method (with theorem arguments and context)"
6.4 Generalized elimination

\textbf{obtain} : \text{proof}(\text{state}) \mid \text{proof}(\text{chain}) \rightarrow \text{proof}(\text{prove})

\textbf{guess}^*: \text{proof}(\text{state}) \mid \text{proof}(\text{chain}) \rightarrow \text{proof}(\text{prove})

Generalized elimination means that additional elements with certain properties may be introduced in the current context, by virtue of a locally proven “soundness statement”. Technically speaking, the \textbf{obtain} language element is like a declaration of \textbf{fix} and \textbf{assume} (see also see §6.2.1), together with a soundness proof of its additional claim. According to the nature of existential reasoning, assumptions get eliminated from any result exported from the context later, provided that the corresponding parameters do not occur in the conclusion.

\[ \langle \text{using } b_1 \ldots b_k \rangle \textbf{obtain } x_1 \ldots x_m \text{ where } a: \varphi_1 \ldots \varphi_n \langle \text{proof} \rangle \equiv \]
\[ \text{have } \bigwedge \text{thesis}. \left( \bigwedge x_1 \ldots x_m. \varphi_1 \implies \ldots \varphi_n \implies \text{thesis} \right) \implies \text{thesis} \]
\[ \text{proof } \text{succeed} \]
\[ \text{fix } \text{thesis} \]
\[ \text{assume that } [\text{Pure.intro?}]: \bigwedge x_1 \ldots x_m. \varphi_1 \implies \ldots \varphi_n \implies \text{thesis} \]
\[ \text{then show } \text{thesis} \]
\[ \text{apply } - \]
\[ \text{using } b_1 \ldots b_k \langle \text{proof} \rangle \]
\[ \text{qed} \]
\[ \text{fix } x_1 \ldots x_m \text{ assume}^*: a: \varphi_1 \ldots \varphi_n \]

Typically, the soundness proof is relatively straight-forward, often just by canonical automated tools such as “by simp” or “by blast”. Accordingly, the “that” reduction above is declared as simplification and introduction rule.
In a sense, \texttt{obtain} represents at the level of Isar proofs what would be meta-
logical existential quantifiers and conjunctions. This concept has a broad
range of useful applications, ranging from plain elimination (or introduction)
of object-level existential and conjunctions, to elimination over results of
symbolic evaluation of recursive definitions, for example. Also note that
\texttt{obtain} without parameters acts much like \texttt{have}, where the result is treated
as a genuine assumption.

An alternative name to be used instead of “that” above may be given in
parentheses.

The improper variant \texttt{guess} is similar to \texttt{obtain}, but derives the obtained
statement from the course of reasoning! The proof starts with a fixed goal
\textit{thesis}. The subsequent proof may refine this to anything of the form like
$\forall x_1 \ldots x_m. \varphi_1 \implies \ldots \varphi_n \implies \text{thesis}$, but must not introduce new subgoals.
The final goal state is then used as reduction rule for the obtain scheme
described above. Obtained parameters $x_1, \ldots, x_m$ are marked as internal
by default, which prevents the proof context from being polluted by ad-hoc
variables. The variable names and type constraints given as arguments for
\texttt{guess} specify a prefix of obtained parameters explicitly in the text.

It is important to note that the facts introduced by \texttt{obtain} and \texttt{guess} may
not be polymorphic: any type-variables occurring here are fixed in the present
context!

\section{Calculational reasoning}

\begin{align*}
\text{also} & : \text{proof(state)} \rightarrow \text{proof(state)} \\
\text{finally} & : \text{proof(state)} \rightarrow \text{proof(chain)} \\
\text{moreover} & : \text{proof(state)} \rightarrow \text{proof(state)} \\
\text{ultimately} & : \text{proof(state)} \rightarrow \text{proof(chain)} \\
\text{print\_trans\_rules}^* & : \text{context} \rightarrow \\
\text{trans} & : \text{attribute} \\
\text{sym} & : \text{attribute} \\
\text{symmetric} & : \text{attribute}
\end{align*}

Calculational proof is forward reasoning with implicit application of transi-
tivity rules (such those of $=, \leq, <$). Isabelle/Isar maintains an auxiliary fact
register \textit{calculation} for accumulating results obtained by transitivity com-
piled with the current result. Command \texttt{also} updates \textit{calculation} involving
\texttt{this}, while \texttt{finally} exhibits the final \textit{calculation} by forward chaining towards
the next goal statement. Both commands require valid current facts, i.e. may
occur only after commands that produce theorems such as \texttt{assume}, \texttt{note}, or some finished proof of \texttt{have}, \texttt{show} etc. The \texttt{moreover} and \texttt{ultimately} commands are similar to \texttt{also} and \texttt{finally}, but only collect further results in \texttt{calculation} without applying any rules yet.

Also note that the implicit term abbreviation “…” has its canonical application with calculational proofs. It refers to the argument of the preceding statement. (The argument of a curried infix expression happens to be its right-hand side.)

Isabelle/Isar calculations are implicitly subject to block structure in the sense that new threads of calculational reasoning are commenced for any new block (as opened by a local goal, for example). This means that, apart from being able to nest calculations, there is no separate \texttt{begin-calculation} command required.

The Isar calculation proof commands may be defined as follows:\textsuperscript{2}

\begin{align*}
\texttt{also}_0 & \equiv \texttt{note} \; \texttt{calculation} = \texttt{this} \\
\texttt{also}_{n+1} & \equiv \texttt{note} \; \texttt{calculation} = \texttt{trans} \; [\texttt{OF} \; \texttt{calculation} \; \texttt{this}] \\
\texttt{finally} & \equiv \texttt{also} \; \texttt{from} \; \texttt{calculation} \\
\texttt{moreover} & \equiv \texttt{note} \; \texttt{calculation} = \texttt{calculation} \; \texttt{this} \\
\texttt{ultimately} & \equiv \texttt{moreover} \; \texttt{from} \; \texttt{calculation}
\end{align*}

\texttt{also} \; (a_1 \ldots a_n) maintains the auxiliary \texttt{calculation} register as follows. The first occurrence of \texttt{also} in some calculational thread initializes \texttt{calculation} by \texttt{this}. Any subsequent \texttt{also} on the same level of block-structure updates \texttt{calculation} by some transitivity rule applied to \texttt{calculation} and \texttt{this} (in that order). Transitivity rules are picked from

\textsuperscript{2}We suppress internal bookkeeping such as proper handling of block-structure.
the current context, unless alternative rules are given as explicit arguments.

finally \((a_1 \ldots a_n)\) maintaining calculation in the same way as also, and concludes the current calculational thread. The final result is exhibited as fact for forward chaining towards the next goal. Basically, finally just abbreviates also from calculation. Typical idioms for concluding calculational proofs are “finally show \(\text{thesis}\)” and “finally have \(\varphi\).”

moreover and ultimately are analogous to also and finally, but collect results only, without applying rules.

print_trans_rules prints the list of transitivity rules (for calculational commands also and finally) and symmetry rules (for the symmetric operation and single step elimination patterns) of the current context.

trans declares theorems as transitivity rules.

sym declares symmetry rules, as well as Pure.elim? rules.

symmetric resolves a theorem with some rule declared as sym in the current context. For example, “assume \([\text{symmetric}]\): \(x = y\)” produces a swapped fact derived from that assumption.

In structured proof texts it is often more appropriate to use an explicit single-step elimination proof, such as “assume \(x = y\) then have \(y = x\) ..”.

6.6 Proof by cases and induction

6.6.1 Rule contexts

\[
\begin{align*}
\text{case} &: \quad \text{proof}(\text{state}) \rightarrow \text{proof}(\text{state}) \\
\text{print_cases}^* &: \quad \text{context} \rightarrow \\
\text{case_names} &: \quad \text{attribute} \\
\text{case_conclusion} &: \quad \text{attribute} \\
\text{params} &: \quad \text{attribute} \\
\text{consumes} &: \quad \text{attribute}
\end{align*}
\]

The puristic way to build up Isar proof contexts is by explicit language elements like fix, assume, let (see §6.2.1). This is adequate for plain natural
deduction, but easily becomes unwieldy in concrete verification tasks, which typically involve big induction rules with several cases.

The `case` command provides a shorthand to refer to a local context symbolically: certain proof methods provide an environment of named “cases” of the form $c: x_1, \ldots, x_m, \varphi_1, \ldots, \varphi_n$; the effect of “case $c$” is then equivalent to “fix $x_1 \ldots x_m$ assume $c: \varphi_1 \ldots \varphi_n$”. Term bindings may be covered as well, notably `?case` for the main conclusion.

By default, the “terminology” $x_1, \ldots, x_m$ of a case value is marked as hidden, i.e. there is no way to refer to such parameters in the subsequent proof text. After all, original rule parameters stem from somewhere outside of the current proof text. By using the explicit form “case $(c \, y_1 \ldots y_m)$” instead, the proof author is able to chose local names that fit nicely into the current context.

It is important to note that proper use of `case` does not provide means to peek at the current goal state, which is not directly observable in Isar! Nonetheless, goal refinement commands do provide named cases `goal_i` for each subgoal $i = 1, \ldots, n$ of the resulting goal state. Using this extra feature requires great care, because some bits of the internal tactical machinery intrude the proof text. In particular, parameter names stemming from the left-over of automated reasoning tools are usually quite unpredictable.

Under normal circumstances, the text of cases emerge from standard elimination or induction rules, which in turn are derived from previous theory specifications in a canonical way (say from `inductive` definitions).

Proper cases are only available if both the proof method and the rules involved support this. By using appropriate attributes, case names, conclusions, and parameters may be also declared by hand. Thus variant versions of rules that have been derived manually become ready to use in advanced case analysis later.
case $(c \, x_1 \ldots x_m)$ invokes a named local context $c: x_1, \ldots, x_m, \varphi_1, \ldots, \varphi_m$, as provided by an appropriate proof method (such as cases and induct). The command “case $(c \, x_1 \ldots x_m)$” abbreviates “fix $x_1 \ldots x_m$ assume $c: \varphi_1 \ldots \varphi_n$”.

print_cases prints all local contexts of the current state, using Isar proof language notation.
case_names $c_1 \ldots c_k$ declares names for the local contexts of premises of a theorem; $c_1, \ldots, c_k$ refers to the prefix of the list of premises. Each of the cases $c_i$ can be of the form $c[h_1 \ldots h_n]$ where the $h_1 \ldots h_n$ are the names of the hypotheses in case $c_i$ from left to right.

case_conclusion $c$ $d_1 \ldots d_k$ declares names for the conclusions of a named premise $c$; here $d_1, \ldots, d_k$ refers to the prefix of arguments of a logical formula built by nesting a binary connective (e.g. $\lor$).

Note that proof methods such as induct and coinduct already provide a default name for the conclusion as a whole. The need to name sub-formulas only arises with cases that split into several sub-cases, as in common co-induction rules.

params $p_1 \ldots p_m$ and $q_1 \ldots q_n$ renames the innermost parameters of premises 1, \ldots, $n$ of some theorem. An empty list of names may be given to skip positions, leaving the present parameters unchanged.

Note that the default usage of case rules does not directly expose parameters to the proof context.

consumes $n$ declares the number of “major premises” of a rule, i.e. the number of facts to be consumed when it is applied by an appropriate proof method. The default value of consumes is $n = 1$, which is appropriate for the usual kind of cases and induction rules for inductive sets (cf. §10.2). Rules without any consumes declaration given are treated as if consumes 0 had been specified.

Note that explicit consumes declarations are only rarely needed; this is already taken care of automatically by the higher-level cases, induct, and coinduct declarations.

### 6.6.2 Proof methods

- **cases** : method
- **induct** : method
- **induction** : method
- **coinduct** : method

The cases, induct, induction, and coinduct methods provide a uniform interface to common proof techniques over datatypes, inductive predicates (or sets), recursive functions etc. The corresponding rules may be specified and instantiated in a casual manner. Furthermore, these methods provide named
local contexts that may be invoked via the case proof command within the subsequent proof text. This accommodates compact proof texts even when reasoning about large specifications.

The induct method also provides some additional infrastructure in order to be applicable to structure statements (either using explicit meta-level connectives, or including facts and parameters separately). This avoids cumbersome encoding of “strengthened” inductive statements within the object-logic.

Method induction differs from induct only in the names of the facts in the local context invoked by the case command.
cases insts R applies method rule with an appropriate case distinction theorem, instantiated to the subjects insts. Symbolic case names are bound according to the rule’s local contexts.
The rule is determined as follows, according to the facts and arguments passed to the cases method:

<table>
<thead>
<tr>
<th>facts</th>
<th>arguments</th>
<th>rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>cases</td>
<td></td>
<td>classical case split</td>
</tr>
<tr>
<td>cases t</td>
<td></td>
<td>datatype exhaustion (type of t)</td>
</tr>
<tr>
<td>⊢ A t cases</td>
<td></td>
<td>inductive predicate/set elimination (of A)</td>
</tr>
<tr>
<td>... cases</td>
<td>... rule: R</td>
<td>explicit rule R</td>
</tr>
</tbody>
</table>

Several instantiations may be given, referring to the suffix of premises of the case rule; within each premise, the prefix of variables is instantiated. In most situations, only a single term needs to be specified; this refers to the first variable of the last premise (it is usually the same for all cases). The (no_simp) option can be used to disable pre-simplification of cases (see the description of induct below for details).

induct insts R and induction insts R are analogous to the cases method, but refer to induction rules, which are determined as follows:

<table>
<thead>
<tr>
<th>facts</th>
<th>arguments</th>
<th>rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>induct P x</td>
<td></td>
<td>datatype induction (type of x)</td>
</tr>
<tr>
<td>⊢ A x induct</td>
<td></td>
<td>predicate/set induction (of A)</td>
</tr>
<tr>
<td>... induct</td>
<td>... rule: R</td>
<td>explicit rule R</td>
</tr>
</tbody>
</table>

Several instantiations may be given, each referring to some part of a mutual inductive definition or datatype — only related partial induction rules may be used together, though. Any of the lists of terms P, x, ... refers to the suffix of variables present in the induction rule. This enables the writer to specify only induction variables, or both predicates and variables, for example.

Instantiations may be definitional: equations \( x \equiv t \) introduce local definitions, which are inserted into the claim and discharged after applying the induction rule. Equalities reappear in the inductive cases, but have been transformed according to the induction principle being involved here. In order to achieve practically useful induction hypotheses, some variables occurring in \( t \) need to be fixed (see below). Instantiations of the form \( t \), where \( t \) is not a variable, are taken as a shorthand for \( x \equiv t \), where \( x \) is a fresh variable. If this is not intended, \( t \) has to be enclosed in parentheses. By default, the equalities generated by definitional instantiations are pre-simplified using a specific set of rules, usually consisting of distinctness and injectivity theorems for datatypes. This pre-simplification may cause some of the parameters of an inductive...
case to disappear, or may even completely delete some of the inductive cases, if one of the equalities occurring in their premises can be simplified to False. The \textit{(no simp)} option can be used to disable pre-simplification. Additional rules to be used in pre-simplification can be declared using the \textit{induct simp} attribute.

The optional “\textit{arbitrary: }x_1 \ldots x_m\textit{” specification generalizes variables }x_1, \ldots, x_m\textit{ of the original goal before applying induction. One can separate variables by “and” to generalize them in other goals then the first. Thus induction hypotheses may become sufficiently general to get the proof through. Together with definitional instantiations, one may effectively perform induction over expressions of a certain structure.

The optional “\textit{taking: }t_1 \ldots t_n\textit{” specification provides additional instantiations of a prefix of pending variables in the rule. Such schematic induction rules rarely occur in practice, though.

\textit{coinduct inst R} is analogous to the \textit{induct} method, but refers to coinduction rules, which are determined as follows:

\begin{table}[h]
\centering
\begin{tabular}{lcl}
\textbf{goal} & \textbf{arguments} & \textbf{rule} \\
\hline
coinduct & \textit{x} & type coinduction (type of \textit{x}) \\
\textit{A \ x} & \textit{coinduct} & \ldots \ predicate/set coinduction (of \textit{A}) \\
\ldots & \textit{coinduct} & \ldots \ rule: \textit{R} \ explicit rule \textit{R}
\end{tabular}
\end{table}

Coinduction is the dual of induction. Induction essentially eliminates }A \ x\text{ towards a generic result }P \ x\text{, while coinduction introduces }A \ x\text{ starting with }B \ x\text{, for a suitable “bisimulation” }B\text{. The cases of a coinduct rule are typically named after the predicates or sets being covered, while the conclusions consist of several alternatives being named after the individual destructor patterns.

The given instantiation refers to the \textit{suffix} of variables occurring in the rule’s major premise, or conclusion if unavailable. An additional “\textit{taking: }t_1 \ldots t_n\textit{” specification may be required in order to specify the bisimulation to be used in the coinduction step.

Above methods produce named local contexts, as determined by the instantiated rule as given in the text. Beyond that, the \textit{induct} and \textit{coinduct} methods guess further instantiations from the goal specification itself. Any persisting unresolved schematic variables of the resulting rule will render the the corresponding case invalid. The term binding \textit{?case} for the conclusion will be provided with each case, provided that term is fully specified.
The \texttt{print\_cases} command prints all named cases present in the current proof state.

Despite the additional infrastructure, both \texttt{cases} and \texttt{coinduct} merely apply a certain rule, after instantiation, while conforming due to the usual way of monotonic natural deduction: the context of a structured statement $\bigwedge x_1 \ldots x_m. \varphi_1 \implies \ldots \varphi_n \implies \ldots$ reappears unchanged after the case split.

The \texttt{induct} method is fundamentally different in this respect: the meta-level structure is passed through the “recursive” course involved in the induction. Thus the original statement is basically replaced by separate copies, corresponding to the induction hypotheses and conclusion; the original goal context is no longer available. Thus local assumptions, fixed parameters and definitions effectively participate in the inductive rephrasing of the original statement.

In \texttt{induct} proofs, local assumptions introduced by cases are split into two different kinds: \texttt{hyps} stemming from the rule and \texttt{prems} from the goal statement. This is reflected in the extracted cases accordingly, so invoking “\texttt{case c}” will provide separate facts $c.\texttt{hyps}$ and $c.\texttt{prems}$, as well as fact $c$ to hold the all-inclusive list.

In \texttt{induction} proofs, local assumptions introduced by cases are split into three different kinds: \texttt{IH}, the induction hypotheses, \texttt{hyps}, the remaining hypotheses stemming from the rule, and \texttt{prems}, the assumptions from the goal statement. The names are $c.\texttt{IH}$, $c.\texttt{hyps}$ and $c.\texttt{prems}$, as above.

Facts presented to either method are consumed according to the number of “major premises” of the rule involved, which is usually 0 for plain cases and induction rules of datatypes etc. and 1 for rules of inductive predicates or sets and the like. The remaining facts are inserted into the goal verbatim before the actual \texttt{cases}, \texttt{induct}, or \texttt{coinduct} rule is applied.

### 6.6.3 Declaring rules

\begin{verbatim}
print_induct_rules* : context →
cases : attribute
induct : attribute
coinduct : attribute
\end{verbatim}
`print_induct_rules` prints cases and induct rules for predicates (or sets) and types of the current context.

`cases`, `induct`, and `coinduct` (as attributes) declare rules for reasoning about (co)inductive predicates (or sets) and types, using the corresponding methods of the same name. Certain definitional packages of object-logics usually declare emerging cases and induction rules as expected, so users rarely need to intervene.

Rules may be deleted via the `del` specification, which covers all of the `type/pred/set` sub-categories simultaneously. For example, `cases del` removes any `cases` rules declared for some type, predicate, or set.

Manual rule declarations usually refer to the `case_names` and `params` attributes to adjust names of cases and parameters of a rule; the `consumes` declaration is taken care of automatically: `consumes 0` is specified for “type” rules and `consumes 1` for “predicate” / “set” rules.
Chapter 7

Inner syntax — the term language

The inner syntax of Isabelle provides concrete notation for the main entities of the logical framework, notably $\lambda$-terms with types and type classes. Applications may either extend existing syntactic categories by additional notation, or define new sub-languages that are linked to the standard term language via some explicit markers. For example $\text{FOO} \ foo$ could embed the syntax corresponding for some user-defined nonterminal $foo$ — within the bounds of the given lexical syntax of Isabelle/Pure.

The most basic way to specify concrete syntax for logical entities works via mixfix annotations (§7.2), which may be usually given as part of the original declaration or via explicit notation commands later on (§7.3). This already covers many needs of concrete syntax without having to understand the full complexity of inner syntax layers.

Further details of the syntax engine involves the classical distinction of lexical language versus context-free grammar (see §7.4), and various mechanisms for syntax translations — either as rewrite systems on first-order ASTs (§7.5) or ML functions on ASTs or $\lambda$-terms that represent parse trees (§7.6).

7.1 Printing logical entities

7.1.1 Diagnostic commands

\[
\begin{align*}
\text{typ}^* : & \quad \text{context} \to \\
\text{term}^* : & \quad \text{context} \to \\
\text{prop}^* : & \quad \text{context} \to \\
\text{thm}^* : & \quad \text{context} \to \\
\text{prf}^* : & \quad \text{context} \to \\
\text{full_prf}^* : & \quad \text{context} \to \\
\text{pr}^* : & \quad \text{any} \to
\end{align*}
\]
These diagnostic commands assist interactive development by printing internal logical entities in a human-readable fashion.

\texttt{typ} \tau reads and prints types of the meta-logic according to the current theory or proof context.
term \( t \) and prop \( \varphi \) read, type-check and print terms or propositions according to the current theory or proof context; the inferred type of \( t \) is output as well. Note that these commands are also useful in inspecting the current environment of term abbreviations.

thm \( a_1 \ldots a_n \) retrieves theorems from the current theory or proof context. Note that any attributes included in the theorem specifications are applied to a temporary context derived from the current theory or proof; the result is discarded, i.e. attributes involved in \( a_1, \ldots, a_n \) do not have any permanent effect.

prf displays the (compact) proof term of the current proof state (if present), or of the given theorems. Note that this requires proof terms to be switched on for the current object logic (see the “Proof terms” section of the Isabelle reference manual for information on how to do this).

full_prf is like prf, but displays the full proof term, i.e. also displays information omitted in the compact proof term, which is denoted by “\_” placeholders there.

pr goals prints the current proof state (if present), including current facts and goals. The optional limit arguments affect the number of goals to be displayed, which is initially 10. Omitting limit value leaves the current setting unchanged.

All of the diagnostic commands above admit a list of modes to be specified, which is appended to the current print mode; see also §7.1.3. Thus the output behavior may be modified according particular print mode features. For example, pr (latex xsymbols) would print the current proof state with mathematical symbols and special characters represented in \LaTeX{} source, according to the Isabelle style [48].

Note that antiquotations (cf. §4.2) provide a more systematic way to include formal items into the printed text document.
7.1.2 Details of printed content

show_types : attribute default false
show_sorts : attribute default false
show_consts : attribute default false
show_abbrves : attribute default true
show_brackets : attribute default false
names_long : attribute default false
names_short : attribute default false
names_unique : attribute default true
eta_contract : attribute default true
goals_limit : attribute default 10
show_main_goal : attribute default false
show_hyps : attribute default false
show_tags : attribute default false
show_question_marks : attribute default true

These configuration options control the detail of information that is displayed for types, terms, theorems, goals etc. See also §9.1.

show_types and show_sorts control printing of type constraints for term variables, and sort constraints for type variables. By default, neither of these are shown in output. If show_sorts is enabled, types are always shown as well.

Note that displaying types and sorts may explain why a polymorphic inference rule fails to resolve with some goal, or why a rewrite rule does not apply as expected.

show_consts controls printing of types of constants when displaying a goal state.

Note that the output can be enormous, because polymorphic constants often occur at several different type instances.

show_abbrves controls folding of constant abbreviations.

show_brackets controls bracketing in pretty printed output. If enabled, all sub-expressions of the pretty printing tree will be parenthesized, even if this produces malformed term syntax! This crude way of showing the internal structure of pretty printed entities may occasionally help to diagnose problems with operator priorities, for example.
names\_long, names\_short, and names\_unique control the way of printing fully qualified internal names in external form. See also §4.2 for the document antiquotation options of the same names.

eta\_contract controls \(\eta\)-contracted printing of terms.

The \(\eta\)-contraction law asserts \((\lambda x. f \, x) \equiv f\), provided \(x\) is not free in \(f\). It asserts extensionality of functions: \(f \equiv g\) if \(f \, x \equiv g \, x\) for all \(x\).

Higher-order unification frequently puts terms into a fully \(\eta\)-expanded form. For example, if \(F\) has type \((\tau \Rightarrow \tau) \Rightarrow \tau\) then its expanded form is \(\lambda h. F \, (\lambda x. h \, x)\).

Enabling eta\_contract makes Isabelle perform \(\eta\)-contractions before printing, so that \(\lambda h. F \, (\lambda x. h \, x)\) appears simply as \(F\).

Note that the distinction between a term and its \(\eta\)-expanded form occasionally matters. While higher-order resolution and rewriting operate modulo \(\alpha\beta\eta\)-conversion, some other tools might look at terms more discretely.

goals\_limit controls the maximum number of subgoals to be shown in goal output.

show\_main\_goal controls whether the main result to be proven should be displayed. This information might be relevant for schematic goals, to inspect the current claim that has been synthesized so far.

show\_hyps controls printing of implicit hypotheses of local facts. Normally, only those hypotheses are displayed that are not covered by the assumptions of the current context: this situation indicates a fault in some tool being used.

By enabling show\_hyps, output of all hypotheses can be enforced, which is occasionally useful for diagnostic purposes.

show\_tags controls printing of extra annotations within theorems, such as internal position information, or the case names being attached by the attribute case\_names.

Note that the tagged and untagged attributes provide low-level access to the collection of tags associated with a theorem.

show\_question\_marks controls printing of question marks for schematic variables, such as \(?x\). Only the leading question mark is affected, the remaining text is unchanged (including proper markup for schematic variables that might be relevant for user interfaces).
7.1.3 Alternative print modes

\[
\text{print\_mode\_value}: \text{unit} \rightarrow \text{string list}
\]

\[
\text{Print\_Mode.with\_modes}: \text{string list} \rightarrow (\text{'a} \rightarrow \text{'b}) \rightarrow \text{'a} \rightarrow \text{'b}
\]

The print mode facility allows to modify various operations for printing. Commands like typ, term, thm (see §7.1.1) take additional print modes as optional argument. The underlying ML operations are as follows.

\[
\text{print\_mode\_value}() \text{ yields the list of currently active print mode names.}
\]

This should be understood as symbolic representation of certain individual features for printing (with precedence from left to right).

\[
\text{Print\_Mode.with\_modes} \text{ modes f x evaluates f x in an execution context where the print mode is prepended by the given modes. This provides a thread-safe way to augment print modes. It is also monotonic in the set of mode names: it retains the default print mode that certain user-interfaces might have installed for their proper functioning!}
\]

The old global reference print\_mode should never be used directly in applications. Its main reason for being publicly accessible is to support historic versions of Proof General.

The pretty printer for inner syntax maintains alternative mixfix productions for any print mode name invented by the user, say in commands like notation or abbreviation. Mode names can be arbitrary, but the following ones have a specific meaning by convention:

- "": (the empty string): default mode; implicitly active as last element in the list of modes.
- input: dummy print mode that is never active; may be used to specify notation that is only available for input.
- internal dummy print mode that is never active; used internally in Isabelle/Pure.
- xsymbols: enable proper mathematical symbols instead of ASCII art.\(^1\)

---

\(^1\)This traditional mode name stems from the “X-Symbol” package for old versions Proof General with XEmacs, although that package has been superseded by Unicode in recent years.
• HTML: additional mode that is active in HTML presentation of Isabelle theory sources; allows to provide alternative output notation.

• latex: additional mode that is active in \LaTeX document preparation of Isabelle theory sources; allows to provide alternative output notation.

7.1.4 Printing limits

Pretty.margin_default: int Unsynchronized.ref
print_depth: int -> unit

These ML functions set limits for pretty printed text.

Pretty.margin_default indicates the global default for the right margin of the built-in pretty printer, with initial value 76. Note that user-interfaces typically control margins automatically when resizing windows, or even bypass the formatting engine of Isabelle/ML altogether and do it within the front end via Isabelle/Scala.

print_depth \( n \) limits the printing depth of the ML toplevel pretty printer; the precise effect depends on the ML compiler and run-time system. Typically \( n \) should be less than 10. Bigger values such as 100–1000 are useful for debugging.

7.2 Mixfix annotations

Mixfix annotations specify concrete inner syntax of Isabelle types and terms. Locally fixed parameters in toplevel theorem statements, locale and class specifications also admit mixfix annotations in a fairly uniform manner. A mixfix annotation describes describes the concrete syntax, the translation to abstract syntax, and the pretty printing. Special case annotations provide a simple means of specifying infix operators and binders.

Isabelle mixfix syntax is inspired by OBJ \[10\]. It allows to specify any context-free priority grammar, which is more general than the fixity declarations of ML and Prolog.

\[
\textit{mixfix}
\]
The string given as `template` may include literal text, spacing, blocks, and arguments (denoted by “_”); the special symbol “\<index>” (printed as “ı”) represents an index argument that specifies an implicit structure reference (see also §5.6). Infix and binder declarations provide common abbreviations for particular mixfix declarations. So in practice, mixfix templates mostly degenerate to literal text for concrete syntax, such as “++” for an infix symbol.
7.2.1 The general mixfix form

In full generality, mixfix declarations work as follows. Suppose a constant $c :: \tau_1 \Rightarrow \ldots \tau_n \Rightarrow \tau$ is annotated by $(\textit{mixfix} \ [p_1, \ldots, p_n] \ p)$, where $\textit{mixfix}$ is a string $d_0 \_ d_1 \_ \ldots \_ d_n$ consisting of delimiters that surround argument positions as indicated by underscores.

Altogether this determines a production for a context-free priority grammar, where for each argument $i$ the syntactic category is determined by $\tau_i$ (with priority $p_i$), and the result category is determined from $\tau$ (with priority $p$). Priority specifications are optional, with default 0 for arguments and 1000 for the result.\(^2\)

Since $\tau$ may be again a function type, the constant type scheme may have more argument positions than the mixfix pattern. Printing a nested application $c \ t_1 \ldots \ t_m$ for $m > n$ works by attaching concrete notation only to the innermost part, essentially by printing $(c \ t_1 \ldots \ t_n) \ldots \ t_m$ instead. If a term has fewer arguments than specified in the mixfix template, the concrete syntax is ignored.

A mixfix template may also contain additional directives for pretty printing, notably spaces, blocks, and breaks. The general template format is a sequence over any of the following entities.

$d$ is a delimiter, namely a non-empty sequence of characters other than the following special characters:

- single quote
- underscore
- index symbol
- open parenthesis
- close parenthesis
- slash

' escapes the special meaning of these meta-characters, producing a literal version of the following character, unless that is a blank.

A single quote followed by a blank separates delimiters, without affecting printing, but input tokens may have additional white space here.

_ is an argument position, which stands for a certain syntactic category in the underlying grammar.

---

\(^2\)Omitting priorities is prone to syntactic ambiguities unless the delimiter tokens determine fully bracketed notation, as in if _ then _ else _ fi.
1 is an indexed argument position; this is the place where implicit structure arguments can be attached.

$s$ is a non-empty sequence of spaces for printing. This and the following specifications do not affect parsing at all.

$(n$ opens a pretty printing block. The optional number specifies how much indentation to add when a line break occurs within the block. If the parenthesis is not followed by digits, the indentation defaults to 0. A block specified via $(00$ is unbreakable.

$)$ closes a pretty printing block.

// forces a line break.

$/s$ allows a line break. Here $s$ stands for the string of spaces (zero or more) right after the slash. These spaces are printed if the break is not taken.

The general idea of pretty printing with blocks and breaks is also described in [35]; it goes back to [29].

### 7.2.2 Infixes

Infix operators are specified by convenient short forms that abbreviate general mixfix annotations as follows:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>infix &quot;sy&quot; $p$</td>
<td>$\mapsto \text{&quot;(_ sy/ _)&quot; [p + 1, p + 1] p}$</td>
<td></td>
</tr>
<tr>
<td>infixl &quot;sy&quot; $p$</td>
<td>$\mapsto \text{&quot;(_ sy/ _)&quot; [p, p + 1] p}$</td>
<td></td>
</tr>
<tr>
<td>infixr &quot;sy&quot; $p$</td>
<td>$\mapsto \text{&quot;(_ sy/ _)&quot; [p + 1, p] p}$</td>
<td></td>
</tr>
</tbody>
</table>

The mixfix template "(_ sy/ _)" specifies two argument positions; the delimiter is preceded by a space and followed by a space or line break; the entire phrase is a pretty printing block.

The alternative notation op sy is introduced in addition. Thus any infix operator may be written in prefix form (as in ML), independently of the number of arguments in the term.

### 7.2.3 Binders

A binder is a variable-binding construct such as a quantifier. The idea to formalize $\forall x. \ b$ as $\text{All} \ (\lambda x. \ b)$ for $\text{All} :: (\forall x \Rightarrow \text{bool}) \Rightarrow \text{bool}$ already goes back to [9]. Isabelle declarations of certain higher-order operators may be annotated with binder annotations as follows:
CHAPTER 7. INNER SYNTAX — THE TERM LANGUAGE

\[ c :: "(\tau_1 \Rightarrow \tau_2) \Rightarrow \tau_3" \quad (\text{binder} "sy" [p] q) \]

This introduces concrete binder syntax \( sy \ x \ . \ b \), where \( x \) is a bound variable of type \( \tau_1 \), the body \( b \) has type \( \tau_2 \) and the whole term has type \( \tau_3 \). The optional integer \( p \) specifies the syntactic priority of the body; the default is \( q \), which is also the priority of the whole construct.

Internally, the binder syntax is expanded to something like this:

\[ c_{\text{binder}} :: "idts \Rightarrow \tau_2 \Rightarrow \tau_3" \quad ("(3sy_./ _)" [0, p] q) \]

Here \( idts \) is the nonterminal symbol for a list of identifiers with optional type constraints (see also §7.4.3). The mixfix template "(3sy_./ _)" defines argument positions for the bound identifiers and the body, separated by a dot with optional line break; the entire phrase is a pretty printing block of indentation level 3. Note that there is no extra space after \( sy \), so it needs to be included user specification if the binder syntax ends with a token that may be continued by an identifier token at the start of \( idts \).

Furthermore, a syntax translation to transforms \( c_{\text{binder}} \ x_1 \ldots \ x_n \ b \) into iterated application \( c (\lambda x_1. \ldots \ c (\lambda x_n. \ b)) \ldots \). This works in both directions, for parsing and printing.

### 7.3 Explicit notation

- **type_notation** : \( \text{local\_theory} \rightarrow \text{local\_theory} \)
- **no_type_notation** : \( \text{local\_theory} \rightarrow \text{local\_theory} \)
- **notation** : \( \text{local\_theory} \rightarrow \text{local\_theory} \)
- **no_notation** : \( \text{local\_theory} \rightarrow \text{local\_theory} \)
- **write** : \( \text{proof}(\text{state}) \rightarrow \text{proof}(\text{state}) \)

Commands that introduce new logical entities (terms or types) usually allow to provide mixfix annotations on the spot, which is convenient for default notation. Nonetheless, the syntax may be modified later on by declarations for explicit notation. This allows to add or delete mixfix annotations for of existing logical entities within the current context.
**type_notation** \( c (mx) \) associates mixfix syntax with an existing type constructor. The arity of the constructor is retrieved from the context.

**no_type_notation** is similar to **type_notation**, but removes the specified syntax annotation from the present context.

**notation** \( c (mx) \) associates mixfix syntax with an existing constant or fixed variable. The type declaration of the given entity is retrieved from the context.

**no_notation** is similar to **notation**, but removes the specified syntax annotation from the present context.

**write** is similar to **notation**, but works within an Isar proof body.
7.4 The Pure syntax

7.4.1 Lexical matters

The inner lexical syntax vaguely resembles the outer one (§3.1), but some details are different. There are two main categories of inner syntax tokens:

1. delimiters — the literal tokens occurring in productions of the given priority grammar (cf. §7.4.2);
2. named tokens — various categories of identifiers etc.

Delimiters override named tokens and may thus render certain identifiers inaccessible. Sometimes the logical context admits alternative ways to refer to the same entity, potentially via qualified names.

The categories for named tokens are defined once and for all as follows, reusing some categories of the outer token syntax (§3.1).

\[
\begin{align*}
  \text{id} &= \text{ident} \\
  \text{longid} &= \text{longident} \\
  \text{var} &= \text{var} \\
  \text{tid} &= \text{typefree} \\
  \text{tvar} &= \text{typevar} \\
  \text{num-token} &= \text{nat} \mid -\text{nat} \\
  \text{float-token} &= \text{nat} \cdot \text{nat} \mid -\text{nat} \cdot \text{nat} \\
  \text{xnum-token} &= \#\text{nat} \mid \#-\text{nat} \\
  \text{str-token} &= ', ' ' '
\end{align*}
\]

The token categories \text{num-token}, \text{float-token}, \text{xnum-token}, and \text{str-token} are not used in Pure. Object-logics may implement numerals and string constants by adding appropriate syntax declarations, together with some translation functions (e.g. see Isabelle/HOL).

The derived categories \text{num-const}, \text{float-const}, and \text{num-const} provide robust access to the respective tokens: the syntax tree holds a syntactic constant instead of a free variable.

7.4.2 Priority grammars

A context-free grammar consists of a set of terminal symbols, a set of non-terminal symbols and a set of productions. Productions have the form \( A = \)
γ, where A is a nonterminal and γ is a string of terminals and nonterminals. One designated nonterminal is called the root symbol. The language defined by the grammar consists of all strings of terminals that can be derived from the root symbol by applying productions as rewrite rules.

The standard Isabelle parser for inner syntax uses a priority grammar. Each nonterminal is decorated by an integer priority: $A^{(p)}$. In a derivation, $A^{(p)}$ may be rewritten using a production $A^{(q)} = \gamma$ only if $p \leq q$. Any priority grammar can be translated into a normal context-free grammar by introducing new nonterminals and productions.

Formally, a set of context free productions $G$ induces a derivation relation $\rightarrow_G$ as follows. Let $\alpha$ and $\beta$ denote strings of terminal or nonterminal symbols. Then $\alpha A^{(p)} \beta \rightarrow_G \alpha \gamma \beta$ holds if and only if $G$ contains some production $A^{(q)} = \gamma$ for $p \leq q$.

The following grammar for arithmetic expressions demonstrates how binding power and associativity of operators can be enforced by priorities.

\[
\begin{align*}
A^{(1000)} &= ( A^{(0)} ) \\
A^{(1000)} &= 0 \\
A^{(0)} &= A^{(0)} + A^{(1)} \\
A^{(2)} &= A^{(3)} * A^{(2)} \\
A^{(3)} &= - A^{(3)}
\end{align*}
\]

The choice of priorities determines that $-$ binds tighter than $\ast$, which binds tighter than $+$. Furthermore $+$ associates to the left and $\ast$ to the right.

For clarity, grammars obey these conventions:

- All priorities must lie between 0 and 1000.
- Priority 0 on the right-hand side and priority 1000 on the left-hand side may be omitted.
- The production $A^{(p)} = \alpha$ is written as $A = \alpha \ (p)$, i.e. the priority of the left-hand side actually appears in a column on the far right.
- Alternatives are separated by $|$.
- Repetition is indicated by dots (\ldots) in an informal but obvious way.

Using these conventions, the example grammar specification above takes the form:
\[
A = (A) \\
\mid 0 \\
\mid A + A^{(1)} (0) \\
\mid A^{(3)} \ast A^{(2)} (2) \\
\mid - A^{(3)} (3)
\]

7.4.3 The Pure grammar

The priority grammar of the Pure theory is defined approximately like this:

\[
\begin{align*}
\text{any} & \quad = \quad \text{prop} \mid \text{logic} \\
\text{prop} & \quad = \quad (\text{prop}) \\
\quad \mid \quad \text{prop}^{(4)} :: \text{type} \quad (3) \\
\quad \mid \quad \text{any}^{(3)} == \text{any}^{(2)} \quad (2) \\
\quad \mid \quad \text{any}^{(3)} \equiv \text{any}^{(2)} \quad (2) \\
\quad \mid \quad \text{prop}^{(3)} \&\& \text{prop}^{(2)} \quad (2) \\
\quad \mid \quad \text{prop}^{(2)} ==> \text{prop}^{(1)} \quad (1) \\
\quad \mid \quad \text{prop}^{(2)} \Longrightarrow \text{prop}^{(1)} \quad (1) \\
\quad \mid \quad [\! [\text{prop} ; \ldots ; \text{prop} ]! ] ==> \text{prop}^{(1)} \quad (1) \\
\quad \mid \quad [\! [\text{prop} ; \ldots ; \text{prop} ]! ] \Longrightarrow \text{prop}^{(1)} \quad (1) \\
\quad \mid \quad ! ! \text{idts} \cdot \text{prop} \quad (0) \\
\quad \mid \quad \text{idts} \cdot \text{prop} \quad (0) \\
\quad \mid \quad \text{OFCLASS} (\text{type}, \text{logic}) \\
\quad \mid \quad \text{SORT_CONSTRAINT} (\text{type}) \\
\quad \mid \quad \text{TERM} \text{logic} \\
\quad \mid \quad \text{PROP} \text{aprop} \\
\text{aprop} & \quad = \quad (\text{aprop}) \\
\quad \mid \quad \text{id} \mid \text{longid} \mid \text{var} \mid - \mid \ldots \\
\quad \mid \quad \text{CONST id} \mid \text{CONST longid} \\
\quad \mid \quad \text{XCONST id} \mid \text{XCONST longid} \\
\quad \mid \quad \text{logic}^{(1000)} \text{any}^{(1000)} \ldots \text{any}^{(1000)} \quad (999) \\
\text{logic} & \quad = \quad (\text{logic}) \\
\quad \mid \quad \text{logic}^{(4)} :: \text{type} \quad (3) \\
\quad \mid \quad \text{id} \mid \text{longid} \mid \text{var} \mid - \mid \ldots
\end{align*}
\]
Here literal terminals are printed verbatim; see also §7.4.1 for further token categories of the inner syntax. The meaning of the nonterminals defined by the above grammar is as follows:
any denotes any term.

prop denotes meta-level propositions, which are terms of type prop. The syntax of such formulae of the meta-logic is carefully distinguished from usual conventions for object-logics. In particular, plain λ-term notation is not recognized as prop.

aprop denotes atomic propositions, which are embedded into regular prop by means of an explicit PROP token.

Terms of type prop with non-constant head, e.g. a plain variable, are printed in this form. Constants that yield type prop are expected to provide their own concrete syntax; otherwise the printed version will appear like logic and cannot be parsed again as prop.

logic denotes arbitrary terms of a logical type, excluding type prop. This is the main syntactic category of object-logic entities, covering plain λ-term notation (variables, abstraction, application), plus anything defined by the user.

When specifying notation for logical entities, all logical types (excluding prop) are collapsed to this single category of logic.

index denotes an optional index term for indexed syntax. If omitted, it refers to the first structure variable in the context. The special dummy “ı” serves as pattern variable in mixfix annotations that introduce indexed notation.

idt denotes identifiers, possibly constrained by types.

idts denotes a sequence of idt. This is the most basic category for variables in iterated binders, such as λ or \( \land \).

pttrn and pttrns denote patterns for abstraction, cases bindings etc. In Pure, these categories start as a merely copy of idt and idts, respectively. Object-logics may add additional productions for binding forms.

type denotes types of the meta-logic.

sort denotes meta-level sorts.

Here are some further explanations of certain syntax features.

- In idts, note that \( x :: \text{nat} \ y \) is parsed as \( x :: (\text{nat} \ y) \), treating \( y \) like a type constructor applied to \( \text{nat} \). To avoid this interpretation, write \( (x :: \text{nat}) \ y \) with explicit parentheses.
• Similarly, \( x :: \text{nat} \ y :: \text{nat} \) is parsed as \( x :: (\text{nat} \ y :: \text{nat}) \). The correct form is \((x :: \text{nat}) \ (y :: \text{nat})\), or \((x :: \text{nat}) \ y :: \text{nat}\) if \(y\) is last in the sequence of identifiers.

• Type constraints for terms bind very weakly. For example, \( x < y :: \text{nat} \) is normally parsed as \((x < y) :: \text{nat}\), unless \(<\) has a very low priority, in which case the input is likely to be ambiguous. The correct form is \(x < (y :: \text{nat})\).

• Constraints may be either written with two literal colons “::” or the double-colon symbol \(<\text{Colon}>\), which actually looks exactly the same in some \LaTeX{} styles.

• Dummy variables (written as underscore) may occur in different roles.

  A type “\(\_\)” or “\(\_ :: \text{sort}\)” acts like an anonymous inference parameter, which is filled-in according to the most general type produced by the type-checking phase.

  A bound “\(\_\)” refers to a vacuous abstraction, where the body does not refer to the binding introduced here. As in the term \(\lambda x \_ \cdot x\), which is \(\alpha\)-equivalent to \(\lambda x y \cdot x\).

  A free “\(\_\)” refers to an implicit outer binding. Higher definitional packages usually allow forms like \(f x \_ = x\).

  A schematic “\(\_\)” (within a term pattern, see §3.2.6) refers to an anonymous variable that is implicitly abstracted over its context of locally bound variables. For example, this allows pattern matching of \(\{x. f x = g x\}\) against \(\{x. \_ = \_\}\), or even \(\{\_ = \_\}\) by using both bound and schematic dummies.

• The three literal dots “\(\ldots\)” may be also written as ellipsis symbol \(<\text{dots}>\). In both cases this refers to a special schematic variable, which is bound in the context. This special term abbreviation works nicely with calculational reasoning (§6.5).

• \texttt{CONST} ensures that the given identifier is treated as constant term, and passed through the parse tree in fully internalized form. This is particularly relevant for translation rules (§7.5), notably on the RHS.

• \texttt{XCONST} is similar to \texttt{CONST}, but retains the constant name as given. This is only relevant to translation rules (§7.5), notably on the LHS.
7.4.4 Inspecting the syntax

print_syntax* : context →

print_syntax prints the inner syntax of the current context. The output can be quite large; the most important sections are explained below.

lexicon lists the delimiters of the inner token language; see §7.4.1.

prods lists the productions of the underlying priority grammar; see §7.4.2.

The nonterminal A^{(p)} is rendered in plain text as A[p]; delimiters are quoted. Many productions have an extra \ldots = \Rightarrow name. These names later become the heads of parse trees; they also guide the pretty printer.

Productions without such parse tree names are called copy productions. Their right-hand side must have exactly one nonterminal symbol (or named token). The parser does not create a new parse tree node for copy productions, but simply returns the parse tree of the right-hand symbol.

If the right-hand side of a copy production consists of a single nonterminal without any delimiters, then it is called a chain production. Chain productions act as abbreviations: conceptually, they are removed from the grammar by adding new productions. Priority information attached to chain productions is ignored; only the dummy value −1 is displayed.

print modes lists the alternative print modes provided by this grammar; see §7.1.3.

parse_rules and print_rules relate to syntax translations (macros); see §7.5.

parse_ast_translation and print_ast_translation list sets of constants that invoke translation functions for abstract syntax trees, which are only required in very special situations; see §7.6.

parse_translation and print_translation list the sets of constants that invoke regular translation functions; see §7.6.

7.4.5 Ambiguity of parsed expressions

syntax_ambiguity_warning : attribute default true

syntax_ambiguity_limit : attribute default 10
Depending on the grammar and the given input, parsing may be ambiguous. Isabelle lets the Earley parser enumerate all possible parse trees, and then tries to make the best out of the situation. Terms that cannot be type-checked are filtered out, which often leads to a unique result in the end. Unlike regular type reconstruction, which is applied to the whole collection of input terms simultaneously, the filtering stage only treats each given term in isolation. Filtering is also not attempted for individual types or raw ASTs (as required for translations).

Certain warning or error messages are printed, depending on the situation and the given configuration options. Parsing ultimately fails, if multiple results remain after the filtering phase.

```plaintext
syntax ambiguity warning controls output of explicit warning messages about syntax ambiguity.

syntax ambiguity limit determines the number of resulting parse trees that are shown as part of the printed message in case of an ambiguity.
```

## 7.5 Raw syntax and translations

```plaintext
nonterminal  : theory → theory
syntax       : theory → theory
no_syntax    : theory → theory
translations : theory → theory
no_translations : theory → theory
```

Unlike mixfix notation for existing formal entities (§7.3), raw syntax declarations provide full access to the priority grammar of the inner syntax. This includes additional syntactic categories (via nonterminal) and free-form grammar productions (via syntax). Additional syntax translations (or macros, via translations) are required to turn resulting parse trees into proper representations of formal entities again.

```
nonterminal
    name
    and
```
**constdecl**

```
constdecl

name :: type
```

**mode**

```
mode

( name output )
```

**transpat**

```
transpat

( nameref )
```

**nonterminal** $c$ declares a type constructor $c$ (without arguments) to act as purely syntactic type: a nonterminal symbol of the inner syntax.
syntax (mode) c :: σ (mx) augments the priority grammar and the pretty printer table for the given print mode (default "). An optional keyword output means that only the pretty printer table is affected.

Following §7.2, the mixfix annotation $mx = \text{template } ps q$ together with type $σ = τ_1 ⇒ \ldots τ_n ⇒ τ$ and specify a grammar production. The template contains delimiter tokens that surround $n$ argument positions ($\_\_\_\_\_\_$). The latter correspond to nonterminal symbols $A_i$ derived from the argument types $τ_i$ as follows:

- prop if $τ_i = \text{prop}$
- logic if $τ_i = (\ldots)κ$ for logical type constructor $κ \neq \text{prop}$
- any if $τ_i = α$ for type variables
- $κ$ if $τ_i = κ$ for nonterminal $κ$ (syntactic type constructor)

Each $A_i$ is decorated by priority $p_i$ from the given list $ps$; misssing priorities default to 0.

The resulting nonterminal of the production is determined similarly from type $τ$, with priority $q$ and default 1000.

Parsing via this production produces parse trees $t_1, \ldots, t_n$ for the argument slots. The resulting parse tree is composed as $c \ t_1 \ldots \ t_n$, by using the syntax constant $c$ of the syntax declaration.

Such syntactic constants are invented on the spot, without formal check wrt. existing declarations. It is conventional to use plain identifiers prefixed by a single underscore (e.g. \texttt{foobar}). Names should be chosen with care, to avoid clashes with unrelated syntax declarations.

The special case of copy production is specified by $c = "$" (empty string). It means that the resulting parse tree $t$ is copied directly, without any further decoration.

no_syntax (mode) decls removes grammar declarations (and translations) resulting from decls, which are interpreted in the same manner as for syntax above.

translations rules specifies syntactic translation rules (i.e. macros): parse / print rules ($\implies$), parse rules ($\rightarrow$), or print rules ($\leftarrow$). Translation patterns may be prefixed by the syntactic category to be used for parsing; the default is logic.
no_translations rules removes syntactic translation rules, which are interpreted in the same manner as for translations above.

Raw syntax and translations provides a slightly more low-level access to the grammar and the form of resulting parse trees. It is often possible to avoid this untyped macro mechanism, and use type-safe abbreviation or notation instead. Some important situations where syntax and translations are really need are as follows:

- Iterated replacement via recursive translations. For example, consider list enumeration \([a, b, c, d]\) as defined in theory List in Isabelle/HOL.

- Change of binding status of variables: anything beyond the built-in binder mixfix annotation requires explicit syntax translations. For example, consider list filter comprehension \([x ← xs . P]\) as defined in theory List in Isabelle/HOL.

### 7.6 Syntax translation functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>parse_ast_translation</td>
<td>theory (\rightarrow) theory</td>
</tr>
<tr>
<td>parse_translation</td>
<td>theory (\rightarrow) theory</td>
</tr>
<tr>
<td>print_translation</td>
<td>theory (\rightarrow) theory</td>
</tr>
<tr>
<td>typed_print_translation</td>
<td>theory (\rightarrow) theory</td>
</tr>
<tr>
<td>print_ast_translation</td>
<td>theory (\rightarrow) theory</td>
</tr>
</tbody>
</table>

Syntax translation functions written in ML admit almost arbitrary manipulations of Isabelle’s inner syntax. Any of the above commands have a single text argument that refers to an ML expression of appropriate type, which are as follows by default:
If the (advanced) option is given, the corresponding translation functions may depend on the current theory or proof context. This allows to implement advanced syntax mechanisms, as translations functions may refer to specific theory declarations or auxiliary proof data.

See also the chapter on “Syntax Transformations” in old [31] for further details on translations on parse trees.
Chapter 8

Other commands

8.1 Inspecting the context

<table>
<thead>
<tr>
<th>Command</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_commands</td>
<td>any →</td>
</tr>
<tr>
<td>print_theory</td>
<td>context →</td>
</tr>
<tr>
<td>print_methods</td>
<td>context →</td>
</tr>
<tr>
<td>print_attributes</td>
<td>context →</td>
</tr>
<tr>
<td>print_theorems</td>
<td>context →</td>
</tr>
<tr>
<td>find_theorems</td>
<td>context →</td>
</tr>
<tr>
<td>find_consts</td>
<td>context →</td>
</tr>
<tr>
<td>thm_deps</td>
<td>context →</td>
</tr>
<tr>
<td>unused_thms</td>
<td>context →</td>
</tr>
<tr>
<td>print_facts</td>
<td>context →</td>
</tr>
<tr>
<td>print_binds</td>
<td>context →</td>
</tr>
</tbody>
</table>

![Diagram of commands and their relationships]

163
CHAPTER 8. OTHER COMMANDS

thmcriterion

find_consts

constcriterion

thm_deps

unused_thms

These commands print certain parts of the theory and proof context. Note that there are some further ones available, such as for the set of rules declared for simplifications.
print\_commands prints Isabelle’s outer theory syntax, including keywords and command.

print\_theory prints the main logical content of the theory context; the “!” option indicates extra verbosity.

print\_methods prints all proof methods available in the current theory context.

print\_attributes prints all attributes available in the current theory context.

print\_theorems prints theorems resulting from the last command; the “!” option indicates extra verbosity.

find\_theorems criteria retrieves facts from the theory or proof context matching all of given search criteria. The criterion name: \(p\) selects all theorems whose fully qualified name matches pattern \(p\), which may contain “*” wildcards. The criteria intro, elim, and dest select theorems that match the current goal as introduction, elimination or destruction rules, respectively. The criterion solves returns all rules that would directly solve the current goal. The criterion simp: \(t\) selects all rewrite rules whose left-hand side matches the given term. The criterion term \(t\) selects all theorems that contain the pattern \(t\) – as usual, patterns may contain occurrences of the dummy “\(\_\)”, schematic variables, and type constraints.

Criteria can be preceded by “!” to select theorems that do not match. Note that giving the empty list of criteria yields all currently known facts. An optional limit for the number of printed facts may be given; the default is 40. By default, duplicates are removed from the search result. Use with\_dups to display duplicates.

find\_consts criteria prints all constants whose type meets all of the given criteria. The criterion strict: \(ty\) is met by any type that matches the type pattern \(ty\). Patterns may contain both the dummy type “\(\_\)” and sort constraints. The criterion ty is similar, but it also matches against subtypes. The criterion name: \(p\) and the prefix “!” function as described for find\_theorems.

thm\_deps \(a_1 \ldots a_n\) visualizes dependencies of facts, using Isabelle’s graph browser tool (see also [48]).
**unused_thms** $A_1 \ldots A_m - B_1 \ldots B_n$ displays all unused theorems in theories $B_1 \ldots B_n$ or their parents, but not in $A_1 \ldots A_m$ or their parents. If $n$ is 0, the end of the range of theories defaults to the current theory. If no range is specified, only the unused theorems in the current theory are displayed.

**print_facts** prints all local facts of the current context, both named and unnamed ones.

**print_binds** prints all term abbreviations present in the context.

### 8.2 System commands

$$
\begin{align*}
\text{cd}^* & : \text{any} \rightarrow \\
\text{pwd}^* & : \text{any} \rightarrow \\
\text{use_thy}^* & : \text{any} \rightarrow \\
\end{align*}
$$

**cd** *path* changes the current directory of the Isabelle process.

**pwd** prints the current working directory.

**use_thy** A preload theory $A$. These system commands are scarcely used when working interactively, since loading of theories is done automatically as required.

Isabelle file specification may contain path variables (e.g. `$ISABELLE_HOME$`) that are expanded accordingly. Note that `~` abbreviates `$USER_HOME$`, and `~~` abbreviates `$ISABELLE_HOME$`. The general syntax for path specifications follows POSIX conventions.
Chapter 9

Generic tools and packages

9.1 Configuration options

Isabelle/Pure maintains a record of named configuration options within the theory or proof context, with values of type \texttt{bool}, \texttt{int}, \texttt{real}, or \texttt{string}. Tools may declare options in ML, and then refer to these values (relative to the context). Thus global reference variables are easily avoided. The user may change the value of a configuration option by means of an associated attribute of the same name. This form of context declaration works particularly well with commands such as \texttt{declare} or \texttt{using} like this:

\begin{verbatim}
declare [[show_main_goal = false]]

notepad
begin
  note [[show_main_goal = true]]
end
\end{verbatim}

For historical reasons, some tools cannot take the full proof context into account and merely refer to the background theory. This is accommodated by configuration options being declared as “global”, which may not be changed within a local context.

\begin{verbatim}
print_configs : context \rightarrow
\end{verbatim}
print_configs prints the available configuration options, with names, types, and current values.

name = value as an attribute expression modifies the named option, with the syntax of the value depending on the option’s type. For bool the default value is true. Any attempt to change a global option in a local context is ignored.

9.2 Basic proof tools

9.2.1 Miscellaneous methods and attributes

unfold : method
fold : method
insert : method
erule\ast : method
derule\ast : method
frule\ast : method
intro : method
elim : method
succeed : method
fail : method
unfold $a_1 \ldots a_n$ and fold $a_1 \ldots a_n$ expand (or fold back) the given definitions throughout all goals; any chained facts provided are inserted into the goal and subject to rewriting as well.

insert $a_1 \ldots a_n$ inserts theorems as facts into all goals of the proof state. Note that current facts indicated for forward chaining are ignored.

erule $a_1 \ldots a_n$, drule $a_1 \ldots a_n$, and frule $a_1 \ldots a_n$ are similar to the basic rule method (see §6.3.3), but apply rules by elim-resolution, destruct-resolution, and forward-resolution, respectively [43]. The optional natural number argument (default 0) specifies additional assumption steps to be performed here.

Note that these methods are improper ones, mainly serving for experimentation and tactic script emulation. Different modes of basic rule application are usually expressed in Isar at the proof language level, rather than via implicit proof state manipulations. For example, a proper single-step elimination would be done using the plain rule method, with forward chaining of current facts.

intro and elim repeatedly refine some goal by intro- or elim-resolution, after having inserted any chained facts. Exactly the rules given as arguments are taken into account; this allows fine-tuned decomposition of a proof problem, in contrast to common automated tools.
succeed yields a single (unchanged) result; it is the identity of the “,” method combinator (cf. §6.3.1).

fail yields an empty result sequence; it is the identity of the “|” method combinator (cf. §6.3.1).

tagged  : attribute
untagged : attribute
THEN   : attribute
COMP   : attribute
unfolded: attribute
folded : attribute
abs_def : attribute
rotated : attribute
elim_format : attribute
standard* : attribute
no_vars* : attribute

tagged name value and untagged name add and remove tags of some theorem. Tags may be any list of string pairs that serve as formal comment. The first string is considered the tag name, the second its value. Note that untagged removes any tags of the same name.
THEN a and COMP a compose rules by resolution. THEN resolves with the first premise of a (an alternative position may be also specified); the COMP version skips the automatic lifting process that is normally intended (cf. RS and COMP in [43]).

unfolded a₁ ... aₙ and folded a₁ ... aₙ expand and fold back again the given definitions throughout a rule.

abs_def turns an equation of the form \( f x y \equiv t \) into \( f \equiv \lambda x y. t \), which ensures that simp or unfold steps always expand it. This also works for object-logic equality.

rotated n rotate the premises of a theorem by n (default 1).

elim_format turns a destruction rule into elimination rule format, by resolving with the rule \( PROP A \implies (PROP A \implies PROP B) \implies PROP B \).

Note that the Classical Reasoner (§9.4) provides its own version of this operation.

standard puts a theorem into the standard form of object-rules at the outermost theory level. Note that this operation violates the local proof context (including active locales).

no_vars replaces schematic variables by free ones; this is mainly for tuning output of pretty printed theorems.

9.2.2 Low-level equational reasoning

subst : method
hypsubst : method
split : method
These methods provide low-level facilities for equational reasoning that are intended for specialized applications only. Normally, single step calculations would be performed in a structured text (see also §6.5), while the Simplifier methods provide the canonical way for automated normalization (see §9.3).

`subst eq` performs a single substitution step using rule `eq`, which may be either a meta or object equality.

`subst (asm) eq` substitutes in an assumption.

`subst (i ... j) eq` performs several substitutions in the conclusion. The numbers `i` to `j` indicate the positions to substitute at. Positions are ordered from the top of the term tree moving down from left to right. For example, in 
\((a + b) + (c + d)\) there are three positions where commutativity of `+` is applicable: 1 refers to `a + b`, 2 to the whole term, and 3 to `c + d`.

If the positions in the list `(i ... j)` are non-overlapping (e.g. `(2 3)` in 
\((a + b) + (c + d)\)) you may assume all substitutions are performed simultaneously. Otherwise the behaviour of `subst` is not specified.

`subst (asm) (i ... j) eq` performs the substitutions in the assumptions. The positions refer to the assumptions in order from left to right. For example, given in a goal of the form 
\(P (a + b) \Longrightarrow P (c + d) \Longrightarrow \ldots\),
position 1 of commutativity of `+` is the subterm `a + b` and position 2 is the subterm `c + d`. 
hypsubst performs substitution using some assumption; this only works for equations of the form \( x = t \) where \( x \) is a free or bound variable.

\texttt{split a1 \ldots an} performs single-step case splitting using the given rules. Splitting is performed in the conclusion or some assumption of the subgoal, depending of the structure of the rule.

Note that the \texttt{simp} method already involves repeated application of split rules as declared in the current context, using \texttt{split}, for example.

\subsection*{9.2.3 Further tactic emulations}

The following improper proof methods emulate traditional tactics. These admit direct access to the goal state, which is normally considered harmful! In particular, this may involve both numbered goal addressing (default 1), and dynamic instantiation within the scope of some subgoal.

Dynamic instantiations refer to universally quantified parameters of a subgoal (the dynamic context) rather than fixed variables and term abbreviations of a (static) Isar context.

Tactic emulation methods, unlike their ML counterparts, admit simultaneous instantiation from both dynamic and static contexts. If names occur in both contexts goal parameters hide locally fixed variables. Likewise, schematic variables refer to term abbreviations, if present in the static context. Otherwise the schematic variable is interpreted as a schematic variable and left to be solved by unification with certain parts of the subgoal.

Note that the tactic emulation proof methods in Isabelle/Isar are consistently named \texttt{foo_tac}. Note also that variable names occurring on left hand sides of instantiations must be preceded by a question mark if they coincide with a keyword or contain dots. This is consistent with the attribute \texttt{where} (see
§6.3.3).

\begin{verbatim}
rule_tac* : method
erule_tac* : method
drule_tac* : method
frule_tac* : method
cut_tac* : method
thin_tac* : method
subgoal_tac* : method
rename_tac* : method
rotate_tac* : method
tactic* : method
raw_tactic* : method
\end{verbatim}
rotate_tac

tactic
dynamic_insts

rule_tac etc. do resolution of rules with explicit instantiation. This works the same way as the ML tactics \texttt{res_inst_tac} etc. (see [43])

Multiple rules may be only given if there is no instantiation; then \texttt{rule_tac} is the same as \texttt{resolve_tac} in ML (see [43]).

cut_tac inserts facts into the proof state as assumption of a subgoal; instantiations may be given as well. Note that the scope of schematic variables is spread over the main goal statement and rule premises are turned into new subgoals. This is in contrast to the regular method \texttt{insert} which inserts closed rule statements.

thin_tac $\varphi$ deletes the specified premise from a subgoal. Note that $\varphi$ may contain schematic variables, to abbreviate the intended proposition; the first matching subgoal premise will be deleted. Removing useless premises from a subgoal increases its readability and can make search tactics run faster.

subgoal_tac $\varphi_1 \ldots \varphi_n$ adds the propositions $\varphi_1 \ldots \varphi_n$ as local premises to a subgoal, and poses the same as new subgoals (in the original context).

rename_tac $x_1 \ldots x_n$ renames parameters of a goal according to the list $x_1, \ldots, x_n$, which refers to the suffix of variables.

rotate_tac $n$ rotates the premises of a subgoal by $n$ positions: from right to left if $n$ is positive, and from left to right if $n$ is negative; the default value is 1.
**tactic text** produces a proof method from any ML text of type **tactic**.
Apart from the usual ML environment and the current proof context, the ML code may refer to the locally bound values **facts**, which indicates any current facts used for forward-chaining.

**raw_tactic** is similar to **tactic**, but presents the goal state in its raw internal form, where simultaneous subgoals appear as conjunction of the logical framework instead of the usual split into several subgoals. While feature this is useful for debugging of complex method definitions, it should not never appear in production theories.

### 9.3 The Simplifier

#### 9.3.1 Simplification methods

```latex
simp : method
simp_all : method
```

![Diagram of simp](image)

```latex
opt
```

```latex
( (no_asm (no_asm_simp (no_asm_use (asm_lr)))))
```
simp invokes the Simplifier, after declaring additional rules according to the arguments given. Note that the only modifier first removes all other rewrite rules, congruences, and looper tactics (including splits), and then behaves like add.

The cong modifiers add or delete Simplifier congruence rules (see also §9.3.3), the default is to add.

The split modifiers add or delete rules for the Splitter (see also [31]), the default is to add. This works only if the Simplifier method has been properly setup to include the Splitter (all major object logics such HOL, HOLCF, FOL, ZF do this already).

simp_all is similar to simp, but acts on all goals (backwards from the last to the first one).

By default the Simplifier methods take local assumptions fully into account, using equational assumptions in the subsequent normalization process, or simplifying assumptions themselves (cf. asm_full_simp_tac in [31]). In structured proofs this is usually quite well behaved in practice: just the local premises of the actual goal are involved, additional facts may be inserted via explicit forward-chaining (via then, from, using etc.).
Additional Simplifier options may be specified to tune the behavior further (mostly for unstructured scripts with many accidental local facts): “(no_asm)” means assumptions are ignored completely (cf. simp_tac), “(no_asm_simp)” means assumptions are used in the simplification of the conclusion but are not themselves simplified (cf. asm_simp_tac), and “(no_asm_use)” means assumptions are simplified but are not used in the simplification of each other or the conclusion (cf. full_simp_tac). For compatibility reasons, there is also an option “(asm_lr)”, which means that an assumption is only used for simplifying assumptions which are to the right of it (cf. asm_lr_simp_tac).

The configuration option depth_limit limits the number of recursive invocations of the simplifier during conditional rewriting.

The Splitter package is usually configured to work as part of the Simplifier. The effect of repeatedly applying split_tac can be simulated by “(simp only: split: a_1 ... a_n)”. There is also a separate split method available for single-step case splitting.

### 9.3.2 Declaring rules

\[
\begin{align*}
\text{print simpset}^* & : \text{context} \rightarrow \text{simp : attribute} \\
& \quad \text{split : attribute} \\
\end{align*}
\]

\begin{center}
\begin{tikzcd}
\text{simp} \ar[dr] \ar[rd] \\
\text{split} \ar[rr,途中] & & \text{add} \\
& & \text{del}
\end{tikzcd}
\end{center}

\text{print simpset} prints the collection of rules declared to the Simplifier, which is also known as “simpset” internally [31].

\text{simpl} declares simplification rules.

\text{split} declares case split rules.

### 9.3.3 Congruence rules

\[
\text{cong} : \text{attribute}
\]
cong declares congruence rules to the Simplifier context.

Congruence rules are equalities of the form

\[ \ldots \implies f \ ?x_1 \ldots \ ?x_n = f \ ?y_1 \ldots \ ?y_n \]

This controls the simplification of the arguments of \( f \). For example, some arguments can be simplified under additional assumptions:

\[ \begin{align*}
?P_1 &\iff ?Q_1 \implies (?Q_1 \implies ?P_2 \iff ?Q_2) \\
(?P_1 \implies ?P_2) &\iff (?Q_1 \implies ?Q_2)
\end{align*} \]

Given this rule, the simplifier assumes \(?Q_1\) and extracts rewrite rules from it when simplifying \(?P_2\). Such local assumptions are effective for rewriting formulae such as \( x = 0 \implies y + x = y \).

The following congruence rule for bounded quantifiers also supplies contextual information — about the bound variable:

\[ \begin{align*}
(?A = ?B) &\implies (\forall x \in ?B \implies ?P x \iff ?Q x) \\
(\forall x \in ?A. ?P x) &\iff (\forall x \in ?B. ?Q x)
\end{align*} \]

This congruence rule for conditional expressions can supply contextual information for simplifying the arms:

\[ \begin{align*}
?p = ?q &\implies (?q \implies ?a = ?c) \implies (\neg ?q \implies ?b = ?d) \\
(if \ ?p \ then \ ?a \ else \ ?b) & = (if \ ?q \ then \ ?c \ else \ ?d)
\end{align*} \]

A congruence rule can also prevent simplification of some arguments. Here is an alternative congruence rule for conditional expressions that conforms to non-strict functional evaluation:

\[ \begin{align*}
?p = ?q &\implies (if \ ?p \ then \ ?a \ else \ ?b) = (if \ ?q \ then \ ?a \ else \ ?b)
\end{align*} \]

Only the first argument is simplified; the others remain unchanged. This can make simplification much faster, but may require an extra case split over the condition \(?q\) to prove the goal.
9.3.4 Simplification procedures

Simplification procedures are ML functions that produce proven rewrite rules on demand. They are associated with higher-order patterns that approximate the left-hand sides of equations. The Simplifier first matches the current redex against one of the LHS patterns; if this succeeds, the corresponding ML function is invoked, passing the Simplifier context and redex term. Thus rules may be specifically fashioned for particular situations, resulting in a more powerful mechanism than term rewriting by a fixed set of rules.

Any successful result needs to be a (possibly conditional) rewrite rule \( t \equiv u \) that is applicable to the current redex. The rule will be applied just as any ordinary rewrite rule. It is expected to be already in \textit{internal form}, bypassing the automatic preprocessing of object-level equivalences.

\[
\text{simproc} \quad : \quad \text{attribute}
\]

\[
\text{simproc_setup} \quad : \quad \text{local\_theory} \to \text{local\_theory}
\]

\textbf{simproc\_setup} defines a named simplification procedure that is invoked by the Simplifier whenever any of the given term patterns match the current redex. The implementation, which is provided as ML source text, needs to be of type \texttt{morphism -> simpset -> cterm -> thm option},
where the `cterm` represents the current redex `r` and the result is supposed to be some proven rewrite rule `r \equiv r'` (or a generalized version), or `NONE` to indicate failure. The `simpset` argument holds the full context of the current Simplifier invocation, including the actual Isar proof context. The `morphism` informs about the difference of the original compilation context wrt. the one of the actual application later on. The optional `identifier` specifies theorems that represent the logical content of the abstract theory of this simproc.

Morphisms and identifiers are only relevant for simprocs that are defined within a local target context, e.g. in a locale.

simproc add: `name` and simproc del: `name` add or delete named simprocs to the current Simplifier context. The default is to add a simproc. Note that `simproc_setup` already adds the new simproc to the subsequent context.

Example

The following simplification procedure for `(?u::unit) = ()` in HOL performs fine-grained control over rule application, beyond higher-order pattern matching. Declaring `unit_eq` as `simp` directly would make the simplifier loop! Note that a version of this simplification procedure is already active in Isabelle/HOL.

```
simproc_setup unit ("x::unit") = {*
  fn _ => fn _ => fn ct =>
    if HOLogic.is_unit (term_of ct) then NONE
    else SOME (mk_meta_eq @{thm unit_eq})
*}
```

Since the Simplifier applies simplification procedures frequently, it is important to make the failure check in ML reasonably fast.

9.3.5 Forward simplification

`simplified : attribute`
opt

\begin{equation}
\begin{array}{c}
(\text{no_asm}) \\
\text{no_asm_simp} \\
\text{no_asm_use}
\end{array}
\end{equation}

simplified $a_1 \ldots a_n$ causes a theorem to be simplified, either by exactly the specified rules $a_1, \ldots, a_n$, or the implicit Simplifier context if no arguments are given. The result is fully simplified by default, including assumptions and conclusion; the options \texttt{no_asm} etc. tune the Simplifier in the same way as the for the \texttt{simp} method.

Note that forward simplification restricts the simplifier to its most basic operation of term rewriting; solver and looper tactics [31] are not involved here. The \texttt{simplified} attribute should be only rarely required under normal circumstances.


9.4 The Classical Reasoner

9.4.1 Basic concepts

Although Isabelle is generic, many users will be working in some extension of classical first-order logic. Isabelle/ZF is built upon theory FOL, while Isabelle/HOL conceptually contains first-order logic as a fragment. Theorem-proving in predicate logic is undecidable, but many automated strategies have been developed to assist in this task.

Isabelle’s classical reasoner is a generic package that accepts certain information about a logic and delivers a suite of automatic proof tools, based on rules that are classified and declared in the context. These proof procedures are slow and simplistic compared with high-end automated theorem provers, but they can save considerable time and effort in practice. They can prove theorems such as Pelletier’s [36] problems 40 and 41 in a few milliseconds (including full proof reconstruction):

\texttt{lemma} $(\exists y. \forall x. F x y \leftrightarrow F x x) \rightarrow \neg (\forall x. \exists y. \forall z. F z y \leftrightarrow \neg F z x)$
\texttt{by blast}

\texttt{lemma} $(\forall z. \exists y. \forall x. f x y \leftrightarrow f x z \land \neg f x x) \rightarrow \neg (\exists z. \forall x. f x z)$
\texttt{by blast}
The proof tools are generic. They are not restricted to first-order logic, and have been heavily used in the development of the Isabelle/HOL library and applications. The tactics can be traced, and their components can be called directly; in this manner, any proof can be viewed interactively.

The sequent calculus

Isabelle supports natural deduction, which is easy to use for interactive proof. But natural deduction does not easily lend itself to automation, and has a bias towards intuitionism. For certain proofs in classical logic, it can not be called natural. The sequent calculus, a generalization of natural deduction, is easier to automate.

A sequent has the form $\Gamma \vdash \Delta$, where $\Gamma$ and $\Delta$ are sets of formulae. The sequent $P_1, \ldots, P_m \vdash Q_1, \ldots, Q_n$ is valid if $P_1 \land \ldots \land P_m$ implies $Q_1 \lor \ldots \lor Q_n$. Thus $P_1, \ldots, P_m$ represent assumptions, each of which is true, while $Q_1, \ldots, Q_n$ represent alternative goals. A sequent is basic if its left and right sides have a common formula, as in $P, Q \vdash Q, R$; basic sequents are trivially valid.

Sequent rules are classified as right or left, indicating which side of the $\vdash$ symbol they operate on. Rules that operate on the right side are analogous to natural deduction’s introduction rules, and left rules are analogous to elimination rules. The sequent calculus analogue of $(\rightarrow I)$ is the rule

$$\frac{P, \Gamma \vdash \Delta, Q}{\Gamma \vdash \Delta, P \rightarrow Q} (\rightarrow R)$$

Applying the rule backwards, this breaks down some implication on the right side of a sequent; $\Gamma$ and $\Delta$ stand for the sets of formulae that are unaffected by the inference. The analogue of the pair $(\lor I_1)$ and $(\lor I_2)$ is the single rule

$$\frac{\Gamma \vdash \Delta, P, Q}{\Gamma \vdash \Delta, P \lor Q} (\lor R)$$

This breaks down some disjunction on the right side, replacing it by both disjuncts. Thus, the sequent calculus is a kind of multiple-conclusion logic. To illustrate the use of multiple formulae on the right, let us prove the classical theorem $(P \rightarrow Q) \lor (Q \rightarrow P)$. Working backwards, we reduce this

\footnote{For first-order logic, sequents can equivalently be made from lists or multisets of formulae.}
CHAPTER 9. GENERIC TOOLS AND PACKAGES

This example is typical of the sequent calculus: start with the desired theorem and apply rules backwards in a fairly arbitrary manner. This yields a surprisingly effective proof procedure. Quantifiers add only few complications, since Isabelle handles parameters and schematic variables. See [35, Chapter 10] for further discussion.

Simulating sequents by natural deduction

Isabelle can represent sequents directly, as in the object-logic LK. But natural deduction is easier to work with, and most object-logics employ it. Fortunately, we can simulate the sequent $P_1, \ldots, P_m \vdash Q_1, \ldots, Q_n$ by the Isabelle formula $P_1 \Rightarrow \ldots \Rightarrow P_m \Rightarrow \neg Q_2 \Rightarrow \ldots \Rightarrow \neg Q_n \Rightarrow Q_1$ where the order of the assumptions and the choice of $Q_1$ are arbitrary. Elim-resolution plays a key role in simulating sequent proofs.

We can easily handle reasoning on the left. Elim-resolution with the rules $(\lor E), (\bot E)$ and $(\exists E)$ achieves a similar effect as the corresponding sequent rules. For the other connectives, we use sequent-style elimination rules instead of destruction rules such as $(\land E1, 2)$ and $(\lor E)$. But note that the rule $(\neg L)$ has no effect under our representation of sequents!

$$\frac{\Gamma \vdash \Delta, P}{\neg P, \Gamma \vdash \Delta} (\neg L)$$

What about reasoning on the right? Introduction rules can only affect the formula in the conclusion, namely $Q_1$. The other right-side formulae are represented as negated assumptions, $\neg Q_2, \ldots, \neg Q_n$. In order to operate on one of these, it must first be exchanged with $Q_1$. Elim-resolution with the swap rule has this effect: $\neg P \Rightarrow (\neg R \Rightarrow P) \Rightarrow R$

To ensure that swaps occur only when necessary, each introduction rule is converted into a swapped form: it is resolved with the second premise of (swap). The swapped form of $(\land I)$, which might be called $(\neg \land E)$, is

$$\neg (P \land Q) \Rightarrow (\neg R \Rightarrow P) \Rightarrow (\neg R \Rightarrow Q) \Rightarrow R$$

Similarly, the swapped form of $(\rightarrow I)$ is

$$\frac{P, Q \vdash Q, P}{P \vdash Q, (Q \rightarrow P)} (\rightarrow R)$$

$$\frac{P \vdash Q, (Q \rightarrow P)}{\vdash (P \rightarrow Q), (Q \rightarrow P)} (\rightarrow R)$$

$$\frac{\vdash (P \rightarrow Q) \lor (Q \rightarrow P)}{(\lor R)}$$

$P, Q \vdash Q, P$

$P \vdash Q, (Q \rightarrow P)$

$(\rightarrow R)$

$(\lor R)$
\[ \neg (P \rightarrow Q) \implies (\neg R \implies P \implies Q) \implies R \]

Swapped introduction rules are applied using elim-resolution, which deletes the negated formula. Our representation of sequents also requires the use of ordinary introduction rules. If we had no regard for readability of intermediate goal states, we could treat the right side more uniformly by representing sequents as

\[ P_1 \implies \ldots \implies P_m \implies \neg Q_1 \implies \ldots \implies \neg Q_n \implies \bot \]

**Extra rules for the sequent calculus**

As mentioned, destruction rules such as \((\land E1, 2)\) and \((\forall E)\) must be replaced by sequent-style elimination rules. In addition, we need rules to embody the classical equivalence between \(P \rightarrow Q\) and \(\neg P \lor Q\). The introduction rules \((\lor I1, 2)\) are replaced by a rule that simulates \((\lor R)\):

\[ (\neg Q \implies P) \implies P \lor Q \]

The destruction rule \((\rightarrow E)\) is replaced by

\[ (P \rightarrow Q) \implies (\neg P \implies R) \implies (Q \implies R) \implies R \]

Quantifier replication also requires special rules. In classical logic, \(\exists x. P x\) is equivalent to \(\neg (\forall x. \neg P x)\); the rules \((\exists R)\) and \((\forall L)\) are dual:

\[
\frac{\Gamma \vdash \Delta, \exists x. P x, P t}{\Gamma \vdash \Delta, \exists x. P x} (\exists R) \quad \frac{P t, \forall x. P x, \Gamma \vdash \Delta}{\forall x. P x, \Gamma \vdash \Delta} (\forall L)
\]

Thus both kinds of quantifier may be replicated. Theorems requiring multiple uses of a universal formula are easy to invent; consider

\[ (\forall x. P x \rightarrow P (f x)) \land P a \rightarrow P (f^n a) \]

for any \(n > 1\). Natural examples of the multiple use of an existential formula are rare; a standard one is \(\exists x. \forall y. P x \rightarrow P y\).

Forgoing quantifier replication loses completeness, but gains decidability, since the search space becomes finite. Many useful theorems can be proved without replication, and the search generally delivers its verdict in a reasonable time. To adopt this approach, represent the sequent rules \((\exists R)\), \((\exists L)\) and \((\forall R)\) by \((\exists I)\), \((\exists E)\) and \((\forall I)\), respectively, and put \((\forall E)\) into elimination form:
CHAPTER 9. GENERIC TOOLS AND PACKAGES

\[ \forall x. \ P x \implies (P t \implies Q) \implies Q \]

Elim-resolution with this rule will delete the universal formula after a single use. To replicate universal quantifiers, replace the rule by

\[ \forall x. \ P x \implies (P t \implies \forall x. \ P x \implies Q) \implies Q \]

To replicate existential quantifiers, replace \((\exists I)\) by

\[ (\neg (\exists x. \ P x) \implies P t) \implies \exists x. \ P x \]

All introduction rules mentioned above are also useful in swapped form. Replication makes the search space infinite; we must apply the rules with care. The classical reasoner distinguishes between safe and unsafe rules, applying the latter only when there is no alternative. Depth-first search may well go down a blind alley; best-first search is better behaved in an infinite search space. However, quantifier replication is too expensive to prove any but the simplest theorems.

9.4.2 Rule declarations

The proof tools of the Classical Reasoner depend on collections of rules declared in the context, which are classified as introduction, elimination or destruction and as safe or unsafe. In general, safe rules can be attempted blindly, while unsafe rules must be used with care. A safe rule must never reduce a provable goal to an unprovable set of subgoals.

The rule \( P \implies P \lor Q \) is unsafe because it reduces \( P \lor Q \) to \( P \), which might turn out as premature choice of an unprovable subgoal. Any rule is unsafe whose premises contain new unknowns. The elimination rule \( \forall x. \ P x \implies (P t \implies Q) \implies Q \) is unsafe, since it is applied via elim-resolution, which discards the assumption \( \forall x. \ P x \) and replaces it by the weaker assumption \( P t \). The rule \( P t \implies \exists x. \ P x \) is unsafe for similar reasons. The quantifier duplication rule \( \forall x. \ P x \implies (P t \implies \forall x. \ P x \implies Q) \implies Q \) is unsafe in a different sense: since it keeps the assumption \( \forall x. \ P x \), it is prone to looping.

In classical first-order logic, all rules are safe except those mentioned above. The safe / unsafe distinction is vague, and may be regarded merely as a way of giving some rules priority over others. One could argue that \((\lor E)\) is unsafe, because repeated application of it could generate exponentially many subgoals. Induction rules are unsafe because inductive proofs are difficult to set up automatically. Any inference is unsafe that instantiates an unknown in the proof state — thus matching must be used, rather than unification.
Even proof by assumption is unsafe if it instantiates unknowns shared with other subgoals.

\[
\text{print_claset} : \text{context} \rightarrow \\
\text{intro} : \text{attribute} \\
\text{elim} : \text{attribute} \\
\text{dest} : \text{attribute} \\
\text{rule} : \text{attribute} \\
\text{iff} : \text{attribute} \\
\text{swapped} : \text{attribute}
\]

\[
\begin{tikzpicture}
  \node (intro) {intro};
  \node (elim) [below right of=intro] {elim};
  \node (dest) [below left of=intro] {dest};
  \node (iff) [below right of=elim] {iff};
  \node (rule) [below left of=iff] {rule};
  \node (del) [below right of=iff] {del};
  \node (add) [below right of=iff] {add};
  \node (nat) [right of=dest] {nat};
  \node (del) [below right of=iff] {del};

  \draw [-latex] (intro) -- (elim);
  \draw [-latex] (intro) -- (rule);
  \draw [-latex] (intro) -- (iff);
  \draw [-latex] (elim) -- (!);
  \draw [-latex] (dest) -- (?);
  \draw [-latex] (iff) -- (add);
  \draw [-latex] (iff) -- (del);
\end{tikzpicture}
\]

\textbf{print_claset} prints the collection of rules declared to the Classical Reasoner, i.e. the \texttt{claset} within the context.

\textit{intro}, \textit{elim}, and \textit{dest} declare introduction, elimination, and destruction rules, respectively. By default, rules are considered as \textit{unsafe} (i.e. not applied blindly without backtracking), while “!” classifies as \textit{safe}. Rule declarations marked by “?” coincide with those of Isabelle/Pure, cf. §6.3.3 (i.e. are only applied in single steps of the \textit{rule} method). The optional natural number specifies an explicit weight argument, which is ignored by the automated reasoning tools, but determines the search order of single rule steps.

Introduction rules are those that can be applied using ordinary resolution. Their swapped forms are generated internally, which will be
applied using elim-resolution. Elimination rules are applied using elim-
resolution. Rules are sorted by the number of new subgoals they will
yield; rules that generate the fewest subgoals will be tried first. Other-
wise, later declarations take precedence over earlier ones.

Rules already present in the context with the same classification are
ignored. A warning is printed if the rule has already been added with
some other classification, but the rule is added anyway as requested.

*rule del* deletes all occurrences of a rule from the classical context, regardless
of its classification as introduction / elimination / destruction and safe / unsafe.

*iff* declares logical equivalences to the Simplifier and the Classical rea-
soner at the same time. Non-conditional rules result in a safe intro-
duction and elimination pair; conditional ones are considered unsafe.
Rules with negative conclusion are automatically inverted (using \( \neg \)-
elimination internally).

The “?!” version of *iff* declares rules to the Isabelle/Pure context only,
and omits the Simplifier declaration.

*swapped* turns an introduction rule into an elimination, by resolving with
the classical swap principle \( \neg P \implies (\neg R \implies P) \implies R \) in the second
position. This is mainly for illustrative purposes: the Classical Reasoner
already swaps rules internally as explained above.

### 9.4.3 Structured methods

```
rule : method
contradiction : method
```

*rule* as offered by the Classical Reasoner is a refinement over the Pure one
(see §6.3.3). Both versions work the same, but the classical version
observes the classical rule context in addition to that of Isabelle/Pure.

Common object logics (HOL, ZF, etc.) declare a rich collection of
classical rules (even if these would qualify as intuitionistic ones), but
only few declarations to the rule context of Isabelle/Pure (§6.3.3).
contradiction solves some goal by contradiction, deriving any result from both \( \neg A \) and \( A \). Chained facts, which are guaranteed to participate, may appear in either order.

9.4.4 Automated methods

- blast : method
- auto : method
- force : method
- fast : method
- slow : method
- best : method
- fastforce : method
- slowsimp : method
- bestsimp : method
- deepen : method
CHAPTER 9. GENERIC TOOLS AND PACKAGES

- fastforce
  - slowsimp
    - bestsimp
  - clasimpmod

- deepen
  - nat
  - clamod

clamod

- intro
- elim
- dest
- del
- !
- ?
- : thmrefs
blast is a separate classical tableau prover that uses the same classical rule declarations as explained before.

Proof search is coded directly in ML using special data structures. A successful proof is then reconstructed using regular Isabelle inferences. It is faster and more powerful than the other classical reasoning tools, but has major limitations too.

- It does not use the classical wrapper tacticals, such as the integration with the Simplifier of fastforce.
- It does not perform higher-order unification, as needed by the
rule \( \forall f \forall x \in \text{range } f \) in HOL. There are often alternatives to such rules, for example \( b = f \; x \Rightarrow b \in \text{range } f \).

- Function variables may only be applied to parameters of the subgoal. (This restriction arises because the prover does not use higher-order unification.) If other function variables are present then the prover will fail with the message Function Var’s argument not a bound variable.

- Its proof strategy is more general than fast but can be slower. If blast fails or seems to be running forever, try fast and the other proof tools described below.

The optional integer argument specifies a bound for the number of unsafe steps used in a proof. By default, blast starts with a bound of 0 and increases it successively to 20. In contrast, (blast lim) tries to prove the goal using a search bound of lim. Sometimes a slow proof using blast can be made much faster by supplying the successful search bound to this proof method instead.

auto combines classical reasoning with simplification. It is intended for situations where there are a lot of mostly trivial subgoals; it proves all the easy ones, leaving the ones it cannot prove. Occasionally, attempting to prove the hard ones may take a long time.

The optional depth arguments in (auto m n) refer to its built-in classical reasoning procedures: \( m \) (default 4) is for blast, which is tried first, and \( n \) (default 2) is for a slower but more general alternative that also takes wrappers into account.

force is intended to prove the first subgoal completely, using many fancy proof tools and performing a rather exhaustive search. As a result, proof attempts may take rather long or diverge easily.

fast, best, slow attempt to prove the first subgoal using sequent-style reasoning as explained before. Unlike blast, they construct proofs directly in Isabelle.

There is a difference in search strategy and back-tracking: fast uses depth-first search and best uses best-first search (guided by a heuristic function: normally the total size of the proof state).

Method slow is like fast, but conducts a broader search: it may, when backtracking from a failed proof attempt, undo even the step of proving a subgoal by assumption.
fastforce, slowsimp, bestsimp are like fast, slow, best, respectively, but use the \textit{Simplifier} as additional wrapper. The name \textit{fastforce}, as opposed to \textit{fastsimp}, reflects the behaviour of this popular method better without requiring an understanding of its implementation.

\textit{deepen} works by exhaustive search up to a certain depth. The start depth is 4 (unless specified explicitly), and the depth is increased iteratively up to 10. Unsafe rules are modified to preserve the formula they act on, so that it be used repeatedly. This method can prove more goals than \textit{fast}, but is much slower, for example if the assumptions have many universal quantifiers.

Any of the above methods support additional modifiers of the context of classical (and simplifier) rules, but the ones related to the \textit{Simplifier} are explicitly prefixed by \textit{simp} here. The semantics of these ad-hoc rule declarations is analogous to the attributes given before. Facts provided by forward chaining are inserted into the goal before commencing proof search.

\textbf{9.4.5 Semi-automated methods}

These proof methods may help in situations when the fully-automated tools fail. The result is a simpler subgoal that can be tackled by other means, such as by manual instantiation of quantifiers.

\begin{verbatim}
safe    :  method
clarify :  method
clarsimp : method
\end{verbatim}

\begin{verbatim}
\textbf{safe} \hspace{2cm} \textbf{clarify} \hspace{2cm} \textbf{clamod}
\hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm}
\textbf{clarsimp} \hspace{2cm} \textbf{clasimpmod}
\end{verbatim}

\textit{safe} repeatedly performs safe steps on all subgoals. It is deterministic, with at most one outcome.
clarify performs a series of safe steps without splitting subgoals; see also clarify_step_tac.

clarsimp acts like clarify, but also does simplification. Note that if the Simplifier context includes a splitter for the premises, the subgoal may still be split.

9.4.6 Single-step tactics

safe_step_tac: Proof.context -> int -> tactic
inst_step_tac: Proof.context -> int -> tactic
step_tac: Proof.context -> int -> tactic
slow_step_tac: Proof.context -> int -> tactic
clarify_step_tac: Proof.context -> int -> tactic

These are the primitive tactics behind the (semi)automated proof methods of the Classical Reasoner. By calling them yourself, you can execute these procedures one step at a time.

safe_step_tac ctx i performs a safe step on subgoal i. The safe wrapper tacticals are applied to a tactic that may include proof by assumption or Modus Ponens (taking care not to instantiate unknowns), or substitution.

inst_step_tac is like safe_step_tac, but allows unknowns to be instantiated.

step_tac ctx i is the basic step of the proof procedure. The unsafe wrapper tacticals are applied to a tactic that tries safe_tac, inst_step_tac, or applies an unsafe rule from the context.

slow_step_tac resembles step_tac, but allows backtracking between using safe rules with instantiation (inst_step_tac) and using unsafe rules. The resulting search space is larger.

clarify_step_tac ctx i performs a safe step on subgoal i. No splitting step is applied; for example, the subgoal $A \land B$ is left as a conjunction. Proof by assumption, Modus Ponens, etc., may be performed provided they do not instantiate unknowns. Assumptions of the form $x = t$ may be eliminated. The safe wrapper tactical is applied.
9.5 Object-logic setup

The very starting point for any Isabelle object-logic is a “truth judgment” that links object-level statements to the meta-logic (with its minimal language of \textit{prop} that covers universal quantification $\land$ and implication $\implies$).

Common object-logics are sufficiently expressive to internalize rule statements over $\land$ and $\implies$ within their own language. This is useful in certain situations where a rule needs to be viewed as an atomic statement from the meta-level perspective, e.g. \( \forall x. x \in A \implies P x \) versus \( \forall x \in A. P x \).

From the following language elements, only the \texttt{atomize} method and \texttt{rule_format} attribute are occasionally required by end-users, the rest is for those who need to setup their own object-logic. In the latter case existing formulations of Isabelle/FOL or Isabelle/HOL may be taken as realistic examples.

Generic tools may refer to the information provided by object-logic declarations internally.

\texttt{judgment} \( c :: \sigma (mx) \) declares constant \( c \) as the truth judgment of the current object-logic. Its type \( \sigma \) should specify a coercion of the category of object-level propositions to \textit{prop} of the Pure meta-logic; the
mixfix annotation (*mx*) would typically just link the object language (internally of syntactic category *logic*) with that of *prop*. Only one judgment declaration may be given in any theory development.

*atomize* (as a method) rewrites any non-atomic premises of a sub-goal, using the meta-level equations declared via *atomize* (as an attribute) beforehand. As a result, heavily nested goals become amenable to fundamental operations such as resolution (cf. the rule method). Giving the “(full)” option here means to turn the whole subgoal into an object-statement (if possible), including the outermost parameters and assumptions as well.

A typical collection of *atomize* rules for a particular object-logic would provide an internalization for each of the connectives of $\land$, $\rightarrow$, and $\equiv$. Meta-level conjunction should be covered as well (this is particularly important for locales, see §5.6).

*rule_format* rewrites a theorem by the equalities declared as *rulify* rules in the current object-logic. By default, the result is fully normalized, including assumptions and conclusions at any depth. The (*no_asm*) option restricts the transformation to the conclusion of a rule.

In common object-logics (HOL, FOL, ZF), the effect of *rule_format* is to replace (bounded) universal quantification ($\forall$) and implication ($\rightarrow$) by the corresponding rule statements over $\land$ and $\rightarrow$. 
Part III

Object-Logics
10.1 Higher-Order Logic

Isabelle/HOL is based on Higher-Order Logic, a polymorphic version of Church’s Simple Theory of Types. HOL can be best understood as a simply-typed version of classical set theory. The logic was first implemented in Gordon’s HOL system [12]. It extends Church’s original logic [9] by explicit type variables (naive polymorphism) and a sound axiomatization scheme for new types based on subsets of existing types.

Andrews’s book [1] is a full description of the original Church-style higher-order logic, with proofs of correctness and completeness wrt. certain set-theoretic interpretations. The particular extensions of Gordon-style HOL are explained semantically in two chapters of the 1993 HOL book [38].

Experience with HOL over decades has demonstrated that higher-order logic is widely applicable in many areas of mathematics and computer science. In a sense, Higher-Order Logic is simpler than First-Order Logic, because there are fewer restrictions and special cases. Note that HOL is weaker than FOL with axioms for ZF set theory, which is traditionally considered the standard foundation of regular mathematics, but for most applications this does not matter. If you prefer ML to Lisp, you will probably prefer HOL to ZF.

The syntax of HOL follows λ-calculus and functional programming. Function application is curried. To apply the function $f$ of type $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ to the arguments $a$ and $b$ in HOL, you simply write $f\ a\ b$ (as in ML or Haskell). There is no “apply” operator; the existing application of the Pure λ-calculus is re-used. Note that in HOL $f\ (a,\ b)$ means “$f$ applied to the pair $(a,\ b)$ (which is notation for $\text{Pair}\ a\ b$). The latter typically introduces extra formal efforts that can be avoided by currying functions by default. Explicit tuples are as infrequent in HOL formalizations as in good ML or Haskell programs.

Isabelle/HOL has a distinct feel, compared to other object-logics like Isabelle/ZF. It identifies object-level types with meta-level types, taking advantage of the default type-inference mechanism of Isabelle/Pure. HOL fully
identifies object-level functions with meta-level functions, with native abstraction and application.

These identifications allow Isabelle to support HOL particularly nicely, but they also mean that HOL requires some sophistication from the user. In particular, an understanding of Hindley-Milner type-inference with type-classes, which are both used extensively in the standard libraries and applications. Beginners can set show_types or even show_sorts to get more explicit information about the result of type-inference.

10.2 Inductive and coinductive definitions

\begin{align*}
\text{inductive} &: \text{local\_theory} \rightarrow \text{local\_theory} \\
\text{inductive\_set} &: \text{local\_theory} \rightarrow \text{local\_theory} \\
\text{coinductive} &: \text{local\_theory} \rightarrow \text{local\_theory} \\
\text{coinductive\_set} &: \text{local\_theory} \rightarrow \text{local\_theory} \\
\text{mono} &: \text{attribute}
\end{align*}

An inductive definition specifies the least predicate or set $R$ closed under given rules: applying a rule to elements of $R$ yields a result within $R$. For example, a structural operational semantics is an inductive definition of an evaluation relation.

Dually, a coinductive definition specifies the greatest predicate or set $R$ that is consistent with given rules: every element of $R$ can be seen as arising by applying a rule to elements of $R$. An important example is using bisimulation relations to formalise equivalence of processes and infinite data structures.

Both inductive and coinductive definitions are based on the Knaster-Tarski fixed-point theorem for complete lattices. The collection of introduction rules given by the user determines a functor on subsets of set-theoretic relations. The required monotonicity of the recursion scheme is proven as a prerequisite to the fixed-point definition and the resulting consequences. This works by pushing inclusion through logical connectives and any other operator that might be wrapped around recursive occurrences of the defined relation: there must be a monotonicity theorem of the form $A \leq B \implies \mathcal{M} A \leq \mathcal{M} B$, for each premise $\mathcal{M} R t$ in an introduction rule. The default rule declarations of Isabelle/HOL already take care of most common situations.
inductive and coinductive define (co)inductive predicates from the introduction rules.

The propositions given as clauses in the where part are either rules of the usual \( \wedge/\implies \) format (with arbitrary nesting), or equalities using \( \equiv \). The latter specifies extra-logical abbreviations in the sense of abbreviation. Introducing abstract syntax simultaneously with the
actual introduction rules is occasionally useful for complex specifications.

The optional for part contains a list of parameters of the (co)inductive predicates that remain fixed throughout the definition, in contrast to arguments of the relation that may vary in each occurrence within the given clauses.

The optional monos declaration contains additional monotonicity theorems, which are required for each operator applied to a recursive set in the introduction rules.

inductive_set and coinductive_set are wrappers for to the previous commands for native HOL predicates. This allows to define (co)inductive sets, where multiple arguments are simulated via tuples.

mono declares monotonicity rules in the context. These rules are involved in the automated monotonicity proof of the above inductive and coinductive definitions.

10.2.1 Derived rules

A (co)inductive definition of $R$ provides the following main theorems:

- $R$.intros is the list of introduction rules as proven theorems, for the recursive predicates (or sets). The rules are also available individually, using the names given them in the theory file;

- $R$.cases is the case analysis (or elimination) rule;

- $R.induct$ or $R.coinduct$ is the (co)induction rule;

- $R.simps$ is the equation unrolling the fixpoint of the predicate one step.

When several predicates $R_1, \ldots, R_n$ are defined simultaneously, the list of introduction rules is called $R_1\_\ldots\_R_n.intro$, the case analysis rules are called $R_1.cases, \ldots, R_n.cases$, and the list of mutual induction rules is called $R_1\_\ldots\_R_n.inducts$. 
10.2.2 Monotonicity theorems

The context maintains a default set of theorems that are used in monotonicity proofs. New rules can be declared via the `mono` attribute. See the main Isabelle/HOL sources for some examples. The general format of such monotonicity theorems is as follows:

- Theorems of the form \( A \leq B \implies \mathcal{M} A \leq \mathcal{M} B \), for proving monotonicity of inductive definitions whose introduction rules have premises involving terms such as \( \mathcal{M} R t \).

- Monotonicity theorems for logical operators, which are of the general form \((\ldots \rightarrow \ldots) \implies \ldots (\ldots \rightarrow \ldots) \implies \ldots \rightarrow \ldots\). For example, in the case of the operator \( \lor \), the corresponding theorem is

\[
\frac{P_1 \rightarrow Q_1 \quad P_2 \rightarrow Q_2}{P_1 \lor P_2 \rightarrow Q_1 \lor Q_2}
\]

- De Morgan style equations for reasoning about the “polarity” of expressions, e.g.

\[
\neg \neg P \leftrightarrow P \quad \neg (P \land Q) \leftrightarrow \neg P \lor \neg Q
\]

- Equations for reducing complex operators to more primitive ones whose monotonicity can easily be proved, e.g.

\[
(P \rightarrow Q) \leftrightarrow \neg P \lor Q \quad \text{Ball } A \ P \equiv \forall x. \ x \in A \rightarrow P \ x
\]

Examples

The finite powerset operator can be defined inductively like this:

```isabelle
inductive_set Fin :: 'a set ⇒ 'a set set for A :: 'a set
where
  empty: {} ∈ Fin A
| insert: a ∈ A ⇒ B ∈ Fin A ⇒ insert a B ∈ Fin A
```

The accessible part of a relation is defined as follows:

```isabelle
inductive acc :: ('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ bool
for r :: 'a ⇒ 'a ⇒ bool (infix "\prec")
where acc: (\\forall y. \ y \prec x ⇒ acc r y) ⇒ acc r x
```
Common logical connectives can be easily characterized as non-recursive inductive definitions with parameters, but without arguments.

\[
\begin{align*}
\text{inductive } & \text{AND for } A \text{ B :: bool} \\
\text{where } & A \implies B \implies \text{AND A B} \\
\text{inductive } & \text{OR for } A \text{ B :: bool} \\
\text{where } & A \implies \text{OR A B} \\
& | B \implies \text{OR A B} \\
\text{inductive } & \text{EXISTS for } B :: 'a \Rightarrow \text{bool} \\
\text{where } & B \ a \implies \text{EXISTS B}
\end{align*}
\]

Here the cases or induct rules produced by the inductive package coincide with the expected elimination rules for Natural Deduction. Already in the original article by Gerhard Gentzen [11] there is a hint that each connective can be characterized by its introductions, and the elimination can be constructed systematically.

### 10.3 Recursive functions

\[
\begin{align*}
\text{primrec} &: \text{local_theory} \rightarrow \text{local_theory} \\
\text{fun} &: \text{local_theory} \rightarrow \text{local_theory} \\
\text{function} &: \text{local_theory} \rightarrow \text{proof(prove)} \\
\text{termination} &: \text{local_theory} \rightarrow \text{proof(prove)}
\end{align*}
\]
primrec defines primitive recursive functions over datatypes (see also \texttt{datatype} and \texttt{rep_datatype}). The given \textit{equations} specify reduction rules that are produced by instantiating the generic combinator for primitive recursion that is available for each datatype.

Each equation needs to be of the form:

\[ f \, x_1 \ldots \, x_m \, (C \, y_1 \ldots \, y_k) \, z_1 \ldots \, z_n = \text{rhs} \]

such that \( C \) is a datatype constructor, \( \text{rhs} \) contains only the free variables on the left-hand side (or from the context), and all recursive occurrences of \( f \) in \( \text{rhs} \) are of the form \( f \ldots \, y_i \ldots \) for some \( i \). At most one reduction rule for each constructor can be given. The order does not matter. For missing constructors, the function is defined to return a default value, but this equation is made difficult to access for users.

The reduction rules are declared as \texttt{simp} by default, which enables standard proof methods like \texttt{simp} and \texttt{auto} to normalize expressions of \( f \) applied to datatype constructions, by simulating symbolic computation via rewriting.
function defines functions by general wellfounded recursion. A detailed
description with examples can be found in [15]. The function is specified
by a set of (possibly conditional) recursive equations with arbitrary
pattern matching. The command generates proof obligations for the
completeness and the compatibility of patterns.

The defined function is considered partial, and the resulting simplifica-
tion rules (named $f$.psimps) and induction rule (named $f$.pinduct) are
guarded by a generated domain predicate $f$.dom. The termination
command can then be used to establish that the function is total.

fun is a shorthand notation for “function (sequential), followed by au-
tomated proof attempts regarding pattern matching and termination.
See [15] for further details.

termination $f$ commences a termination proof for the previously defined
function $f$. If this is omitted, the command refers to the most recent
function definition. After the proof is closed, the recursive equations
and the induction principle is established.

Recursive definitions introduced by the function command accommodate
reasoning by induction (cf. induct): rule $f$.induct refers to a specific induction
rule, with parameters named according to the user-specified equations. Cases
are numbered starting from 1. For primrec, the induction principle coincides
with structural recursion on the datatype where the recursion is carried out.
The equations provided by these packages may be referred later as theo-
rem list $f$.simps, where $f$ is the (collective) name of the functions defined.
Individual equations may be named explicitly as well.

The function command accepts the following options.

sequential enables a preprocessor which disambiguates overlapping patterns
by making them mutually disjoint. Earlier equations take precedence
over later ones. This allows to give the specification in a format very
similar to functional programming. Note that the resulting simplifica-
tion and induction rules correspond to the transformed specification,
not the one given originally. This usually means that each equation
given by the user may result in several theorems. Also note that this
automatic transformation only works for ML-style datatype patterns.

domintros enables the automated generation of introduction rules for the
domain predicate. While mostly not needed, they can be helpful in
some proofs about partial functions.
Example: evaluation of expressions

Subsequently, we define mutual datatypes for arithmetic and boolean expressions, and use \texttt{primrec} for evaluation functions that follow the same recursive structure.

\begin{verbatim}
datatype 'a aexp =
  IF 'a bexp 'a aexp 'a aexp
| Sum 'a aexp 'a aexp
| Diff 'a aexp 'a aexp
| Var 'a
| Num nat
and 'a bexp =
  Less 'a aexp 'a aexp
| And 'a bexp 'a bexp
| Neg 'a bexp
\end{verbatim}

Evaluation of arithmetic and boolean expressions

\begin{verbatim}
primrec evala :: ('a ⇒ nat) ⇒ 'a aexp ⇒ nat
and evalb :: ('a ⇒ nat) ⇒ 'a bexp ⇒ bool
where
  evala env (IF b a1 a2) = (if evalb env b then evala env a1 else evala env a2)
| evala env (Sum a1 a2) = evala env a1 + evala env a2
| evala env (Diff a1 a2) = evala env a1 − evala env a2
| evala env (Var v) = env v
| evala env (Num n) = n
| evalb env (Less a1 a2) = (evala env a1 < evala env a2)
| evalb env (And b1 b2) = (evalb env b1 ∧ evalb env b2)
| evalb env (Neg b) = (¬ evalb env b)
\end{verbatim}

Since the value of an expression depends on the value of its variables, the functions \texttt{evala} and \texttt{evalb} take an additional parameter, an \textit{environment} that maps variables to their values.

Substitution on expressions can be defined similarly. The mapping \( f \) of type \( 'a ⇒ 'a aexp \) given as a parameter is lifted canonically on the types \( 'a aexp \) and \( 'a bexp \), respectively.

\begin{verbatim}
primrec substa :: ('a ⇒ 'b aexp) ⇒ 'a aexp ⇒ 'b aexp
and substb :: ('a ⇒ 'b aexp) ⇒ 'a bexp ⇒ 'b bexp
where
  substa f (IF b a1 a2) = IF (substb f b) (substa f a1) (substa f a2)
| substa f (Sum a1 a2) = Sum (substa f a1) (substa f a2)
| substa f (Diff a1 a2) = Diff (substa f a1) (substa f a2)
| substa f (Var v) = f v
\end{verbatim}
In textbooks about semantics one often finds substitution theorems, which express the relationship between substitution and evaluation. For \( \mathit{\text{a exp}} \) and \( \mathit{\text{b exp}} \), we can prove such a theorem by mutual induction, followed by simplification.

**Lemma subst_one:**
\[
\begin{align*}
\text{evala env} (\text{substa} (\text{Var} (v := a'))) a &= \text{evala (env (v := evala env a'))} a \\
\text{evalb env} (\text{substb} (\text{Var} (v := a'))) b &= \text{evalb (env (v := evala env a'))} b \\
\text{by (induct } a \text{ and } b) \text{ simp_all}
\end{align*}
\]

**Lemma subst_all:**
\[
\begin{align*}
\text{evala env} (\text{substa} s a) &= \text{evala (λx. evala env (s x))} a \\
\text{evalb env} (\text{substb} s b) &= \text{evalb (λx. evala env (s x))} b \\
\text{by (induct } a \text{ and } b) \text{ simp_all}
\end{align*}
\]

**Example: a substitution function for terms**

Functions on datatypes with nested recursion are also defined by mutual primitive recursion.

**Datatype** \( ('a, 'b) \text{ term} = \text{Var } 'a \mid \text{App } 'b (('a, 'b) \text{ term list}) \)

A substitution function on type \( ('a, 'b) \text{ term} \) can be defined as follows, by working simultaneously on \( ('a, 'b) \text{ term list} \):

**primrec subst_term :: ('a ⇒ ('a, 'b) \text{ term}) ⇒ ('a, 'b) \text{ term} ⇒ ('a, 'b) \text{ term} and subst_term_list :: ('a ⇒ ('a, 'b) \text{ term}) ⇒ ('a, 'b) \text{ term list} ⇒ ('a, 'b) \text{ term list}**

**where**
\[
\begin{align*}
\text{subst_term} f (\text{Var } a) &= f a \\
\text{subst_term} f (\text{App } b ts) &= \text{App } b (\text{subst_term_list} f ts) \\
\text{subst_term_list} f [] &= [] \\
\text{subst_term_list} f (t # ts) &= \text{subst_term} f t # \text{subst_term_list} f ts
\end{align*}
\]

The recursion scheme follows the structure of the unfolded definition of type \( ('a, 'b) \text{ term} \). To prove properties of this substitution function, mutual induction is needed:

**Lemma subst_term (subst_term f1 o f2) t = subst_term f1 (subst_term f2 t) and subst_term_list (subst_term f1 o f2) ts = subst_term_list f1 (subst_term_list f2 ts)**

**by (induct } t \text{ and } ts) \text{ simp_all}**
Example: a map function for infinitely branching trees

Defining functions on infinitely branching datatypes by primitive recursion is just as easy.

datatype 'a tree = Atom 'a | Branch nat ⇒ 'a tree

primrec map_tree :: ('a ⇒ 'b) ⇒ 'a tree ⇒ 'b tree
where
  map_tree f (Atom a) = Atom (f a)
| map_tree f (Branch ts) = Branch (λx. map_tree f (ts x))

Note that all occurrences of functions such as ts above must be applied to an argument. In particular, map_tree f ∘ ts is not allowed here.

Here is a simple composition lemma for map_tree:

lemma map_tree g (map_tree f t) = map_tree (g ∘ f) t
  by (induct t) simp_all

10.3.1 Proof methods related to recursive definitions

pat_completeness : method
relation : method
lexicographic_order : method
size_change : method
induction_schema : method
pat_completeness is a specialized method to solve goals regarding the completeness of pattern matching, as required by the function package (cf. [15]).

relation $R$ introduces a termination proof using the relation $R$. The resulting proof state will contain goals expressing that $R$ is wellfounded, and that the arguments of recursive calls decrease with respect to $R$. Usually, this method is used as the initial proof step of manual termination proofs.

lexicographic_order attempts a fully automated termination proof by searching for a lexicographic combination of size measures on the arguments of the function. The method accepts the same arguments as the auto method, which it uses internally to prove local descents. The clasimpmod modifiers are accepted (as for auto).

In case of failure, extensive information is printed, which can help to analyse the situation (cf. [15]).

size_change also works on termination goals, using a variation of the size-change principle, together with a graph decomposition technique (see [16] for details). Three kinds of orders are used internally: $\max$, $\min$, and $\ms$ (multiset), which is only available when the theory Multiset is loaded. When no order kinds are given, they are tried in order. The search for a termination proof uses SAT solving internally.

For local descent proofs, the clasimpmod modifiers are accepted (as for auto).

induction_schema derives user-specified induction rules from well-founded induction and completeness of patterns. This factors out some operations that are done internally by the function package and makes them available separately. See ~/src/HOL/ex/Induction_Schema.thy for examples.
10.3.2 Functions with explicit partiality

```
partial_function : local_theory → local_theory
partial_function_mono : attribute
```

`partial_function` *(mode)* defines recursive functions based on fixpoints in complete partial orders. No termination proof is required from the user or constructed internally. Instead, the possibility of non-termination is modelled explicitly in the result type, which contains an explicit bottom element.

Pattern matching and mutual recursion are currently not supported. Thus, the specification consists of a single function described by a single recursive equation.

There are no fixed syntactic restrictions on the body of the function, but the induced functional must be provably monotonic wrt. the underlying order. The monotonicity proof is performed internally, and the definition is rejected when it fails. The proof can be influenced by declaring hints using the `partial_function_mono` attribute.

The mandatory *mode* argument specifies the mode of operation of the command, which directly corresponds to a complete partial order on the result type. By default, the following modes are defined:

- `option` defines functions that map into the `option` type. Here, the value `None` is used to model a non-terminating computation. Monotonicity requires that if `None` is returned by a recursive call, then the overall result must also be `None`. This is best achieved through the use of the monadic operator `Option.bind`.

- `tailrec` defines functions with an arbitrary result type and uses the slightly degenerated partial order where `undefined` is the bottom
element. Now, monotonicity requires that if \textit{undefined} is returned by a recursive call, then the overall result must also be \textit{undefined}. In practice, this is only satisfied when each recursive call is a tail call, whose result is directly returned. Thus, this mode of operation allows the definition of arbitrary tail-recursive functions.

Experienced users may define new modes by instantiating the locale \textit{partial\_function\_definitions} appropriately.

\textit{partial\_function\_mono} declares rules for use in the internal monononicity proofs of partial function definitions.

### 10.3.3 Old-style recursive function definitions (TFL)

\texttt{recdef} : \textit{theory} \rightarrow \textit{theory}
\texttt{recdef\_tc} : \textit{theory} \rightarrow \textit{proof(prove)}

The old TFL commands \texttt{recdef} and \texttt{recdef\_tc} for defining recursive are mostly obsolete; \texttt{function} or \texttt{fun} should be used instead.
CHAPTER 10. ISABELLE/HOL

recdefmod

\[
\begin{array}{c}
\text{recdef} \mod \\
\text{thmrefs}
\end{array}
\]

\[
\begin{array}{c}
\text{recdef simp} \\
\text{recdef cong} \\
\text{recdef wf}
\end{array}
\]

\[
\begin{array}{c}
\text{add} \\
\text{del}
\end{array}
\]

clasimpmod

tc

\[
\begin{array}{c}
\text{nameref}
\end{array}
\]

\[
\begin{array}{c}
\text{(nat)}
\end{array}
\]

\textbf{recdef} defines general well-founded recursive functions (using the TFL package), see also [24]. The “(permissive)” option tells TFL to recover from failed proof attempts, returning unfinished results. The \texttt{recdef simp}, \texttt{recdef cong}, and \texttt{recdef wf} hints refer to auxiliary rules to be used in the internal automated proof process of TFL. Additional \texttt{clasimpmod} declarations may be given to tune the context of the Simplifier (cf. §9.3) and Classical reasoner (cf. §9.4).

\textbf{recdef tc} \(c\) \((i)\) recommences the proof for leftover termination condition number \(i\) (default 1) as generated by a \texttt{recdef} definition of constant \(c\).

Note that in most cases, \texttt{recdef} is able to finish its internal proofs without manual intervention.

Hints for \texttt{recdef} may be also declared globally, using the following attributes.

\texttt{recdef simp : attribute}
\texttt{recdef cong : attribute}
\texttt{recdef wf : attribute}
10.4 Datatypes

 datatype : theory → theory
 repdatatype : theory → proof(prove)

 datatype defines inductive datatypes in HOL.

 repdatatype represents existing types as datatypes.

 For foundational reasons, some basic types such as nat, 'a × 'b, 'a + 'b, bool and unit are introduced by more primitive means using typedef.
 To recover the rich infrastructure of datatype (e.g. rules for cases and induct and the primitive recursion combinators), such types may be represented as actual datatypes later. This is done by specifying the constructors of the desired type, and giving a proof of the induction rule, distinctness and injectivity of constructors.

 For example, see ~/src/HOL/Sum_Type.thy for the representation of the primitive sum type as fully-featured datatype.
The generated rules for \texttt{induct} and \texttt{cases} provide case names according to the given constructors, while parameters are named after the types (see also §6.6).

See [24] for more details on datatypes, but beware of the old-style theory syntax being used there! Apart from proper proof methods for case-analysis and induction, there are also emulations of ML tactics \texttt{case_tac} and \texttt{induct_tac} available, see §10.19; these admit to refer directly to the internal structure of subgoals (including internally bound parameters).

Examples

We define a type of finite sequences, with slightly different names than the existing \texttt{'a list} that is already in \texttt{Main}:

\begin{verbatim}
datatype 'a seq = Empty | Seq 'a 'a seq
\end{verbatim}

We can now prove some simple lemma by structural induction:

\begin{verbatim}
lemma Seq x xs \neq xs
proof (induct xs arbitrary: x)
  case Empty
  show Seq x Empty \neq Empty by simp
next
  case (Seq y ys)
  show Seq x (Seq y ys) \neq Seq y ys using 'Seq y ys \neq ys' by simp
qed
\end{verbatim}

Here is a more succinct version of the same proof:

\begin{verbatim}
lemma Seq x xs \neq xs
  by (induct xs arbitrary: x) simp_all
\end{verbatim}

10.5 Records

In principle, records merely generalize the concept of tuples, where components may be addressed by labels instead of just position. The logical infrastructure of records in Isabelle/HOL is slightly more advanced, though,
supporting truly extensible record schemes. This admits operations that are polymorphic with respect to record extension, yielding “object-oriented” effects like (single) inheritance. See also [22] for more details on object-oriented verification and record subtyping in HOL.

10.5.1 Basic concepts

Isabelle/HOL supports both fixed and schematic records at the level of terms and types. The notation is as follows:

<table>
<thead>
<tr>
<th>fixed</th>
<th>record terms</th>
<th>record types</th>
</tr>
</thead>
<tbody>
<tr>
<td>schematic</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The ASCII representation of $\{x = a\}$ is $\{x = a\}$.

A fixed record $\{x = a, y = b\}$ has field $x$ of value $a$ and field $y$ of value $b$.

A record scheme like $\{x :: A, y :: B\}$, assuming that $a :: A$ and $b :: B$.

A record scheme like $\{x :: A, y :: B, \ldots :: M\}$ contains fields $x$ and $y$ as before, but also possibly further fields as indicated by the “…” notation (which is actually part of the syntax). The improper field “…” of a record scheme is called the more part. Logically it is just a free variable, which is occasionally referred to as “row variable” in the literature. The more part of a record scheme may be instantiated by zero or more further components. For example, the previous scheme may get instantiated to $\{x = a, y = b, z = c, \ldots = m\}$, where $m'$ refers to a different more part. Fixed records are special instances of record schemes, where “…” is properly terminated by the $() :: unit$ element. In fact, $\{x = a, y = b\}$ is just an abbreviation for $\{x = a, y = b, \ldots = ()\}$.

Two key observations make extensible records in a simply typed language like HOL work out:

1. the more part is internalized, as a free term or type variable,
2. field names are externalized, they cannot be accessed within the logic as first-class values.

In Isabelle/HOL record types have to be defined explicitly, fixing their field names and types, and their (optional) parent record. Afterwards, records may be formed using above syntax, while obeying the canonical order of fields as
given by their declaration. The record package provides several standard operations like selectors and updates. The common setup for various generic proof tools enable succinct reasoning patterns. See also the Isabelle/HOL tutorial [25] for further instructions on using records in practice.

10.5.2 Record specifications

\texttt{record} : \texttt{theory} \rightarrow \texttt{theory}

\texttt{record} (\alpha_1, \ldots, \alpha_m) \ t = \tau + c_1 :: \sigma_1 \ldots c_n :: \sigma_n \text{ defines extensible record type } (\alpha_1, \ldots, \alpha_m) \ t, \text{ derived from the optional parent record } \tau \text{ by adding new field components } c_i :: \sigma_i \text{ etc.}

The type variables of \( \tau \) and \( \sigma_i \) need to be covered by the (distinct) parameters \( \alpha_1, \ldots, \alpha_m \). Type constructor \( t \) has to be new, while \( \tau \) needs to specify an instance of an existing record type. At least one new field \( c_i \) has to be specified. Basically, field names need to belong to a unique record. This is not a real restriction in practice, since fields are qualified by the record name internally.

The parent record specification \( \tau \) is optional; if omitted \( t \) becomes a root record. The hierarchy of all records declared within a theory context forms a forest structure, i.e. a set of trees starting with a root record each. There is no way to merge multiple parent records!

For convenience, \( (\alpha_1, \ldots, \alpha_m) \ t \) is made a type abbreviation for the fixed record type \( (c_1 :: \sigma_1, \ldots, c_n :: \sigma_n) \), likewise is \( (\alpha_1, \ldots, \alpha_m, \zeta) \ t_{\text{scheme}} \) made an abbreviation for \( (c_1 :: \sigma_1, \ldots, c_n :: \sigma_n, \ldots :: \zeta) \).
10.5.3 Record operations

Any record definition of the form presented above produces certain standard operations. Selectors and updates are provided for any field, including the improper one "more". There are also cumulative record constructor functions. To simplify the presentation below, we assume for now that \((\alpha_1, \ldots, \alpha_m) \ t\) is a root record with fields \(c_1 :: \sigma_1, \ldots, c_n :: \sigma_n\).

Selectors and updates are available for any field (including "more"):  
\[
c_i :: (\| \ |
\begin{array}{c}
  c :: \sigma,
  \ldots,
  \zeta:: \zeta
\end{array}
\) \Rightarrow \sigma_i
\]
\[
c_i.update :: \sigma_i \Rightarrow (\| \ |
\begin{array}{c}
  c :: \sigma,
  \ldots,
  \zeta:: \zeta
\end{array}
\) \Rightarrow (\| \ |
\begin{array}{c}
  c :: \sigma,
  \ldots,
  \zeta:: \zeta
\end{array}
\)
\]

There is special syntax for application of updates: \(r(x := a)\) abbreviates term \(_{x.update} a r\). Further notation for repeated updates is also available: \(r(x := a)(y := b)(z := c)\) may be written \(r(x := a, y := b, z := c)\). Note that because of postfix notation the order of fields shown here is reverse than in the actual term. Since repeated updates are just function applications, fields may be freely permuted in \(r(x := a, y := b, z := c)\), as far as logical equality is concerned. Thus commutativity of independent updates can be proven within the logic for any two fields, but not as a general theorem.

The \texttt{make} operation provides a cumulative record constructor function:
\[
t.make :: \sigma_1 \Rightarrow \ldots \sigma_n \Rightarrow (| \| \ |
\begin{array}{c}
  \sigma :: \sigma
\end{array}
\)
\]

We now reconsider the case of non-root records, which are derived of some parent. In general, the latter may depend on another parent as well, resulting in a list of ancestor records. Appending the lists of fields of all ancestors results in a certain field prefix. The record package automatically takes care of this by lifting operations over this context of ancestor fields. Assuming that \((\alpha_1, \ldots, \alpha_m) \ t\) has ancestor fields \(b_1 :: \rho_1, \ldots, b_k :: \rho_k\), the above record operations will get the following types:
\[
c_i :: (\| \ |
\begin{array}{c}
  \bar{b} :: \rho,
  \bar{c} :: \sigma,
  \ldots :: \zeta
\end{array}
\) \Rightarrow \sigma_i
\]
\[
c_i.update :: \sigma_i \Rightarrow (\| \ |
\begin{array}{c}
  \bar{b} :: \rho,
  \bar{c} :: \sigma,
  \ldots :: \zeta
\end{array}
\) \Rightarrow (\| \ |
\begin{array}{c}
  \bar{b} :: \rho,
  \bar{c} :: \sigma,
  \ldots :: \zeta
\end{array}
\)
\]
\[
t.make :: \rho_1 \Rightarrow \ldots \rho_k \Rightarrow \sigma_1 \Rightarrow \ldots \sigma_n \Rightarrow (| \| \ |
\begin{array}{c}
  \bar{b} :: \rho
\end{array}
\)
\]

Some further operations address the extension aspect of a derived record scheme specifically: \(t.fields\) produces a record fragment consisting of exactly the new fields introduced here (the result may serve as a more part elsewhere); \(t.extend\) takes a fixed record and adds a given more part; \(t.truncate\) restricts a record scheme to a fixed record.
\[ t.\text{fields} \quad :: \quad \sigma_1 \Rightarrow \ldots \sigma_n \Rightarrow (\tau :: \sigma) \\\n\]
\[ t.\text{extend} \quad :: \quad (\bar{b} :: \bar{\nu}, \tau :: \sigma) \Rightarrow \zeta \Rightarrow (\bar{b} :: \bar{\nu}, \tau :: \sigma, \ldots :: \zeta) \\\n\]
\[ t.\text{truncate} \quad :: \quad (\bar{b} :: \bar{\nu}, \tau :: \sigma, \ldots :: \zeta) \Rightarrow (\bar{b} :: \bar{\nu}, \tau :: \sigma) \]

Note that \( t.\text{make} \) and \( t.\text{fields} \) coincide for root records.

### 10.5.4 Derived rules and proof tools

The record package proves several results internally, declaring these facts to appropriate proof tools. This enables users to reason about record structures quite conveniently. Assume that \( t \) is a record type as specified above.

1. Standard conversions for selectors or updates applied to record constructor terms are made part of the default Simplifier context; thus proofs by reduction of basic operations merely require the \textit{simp} method without further arguments. These rules are available as \( t.\text{simps} \), too.

2. Selectors applied to updated records are automatically reduced by an internal simplification procedure, which is also part of the standard Simplifier setup.

3. Inject equations of a form analogous to \((x, y) = (x', y') \equiv x = x' \land y = y'\) are declared to the Simplifier and Classical Reasoner as \textit{iff} rules. These rules are available as \( t.\text{iffs} \).

4. The introduction rule for record equality analogous to \( x \ r = x' \ r' \Longrightarrow y \ r = y' \ r' \ldots \Longrightarrow r = r' \) is declared to the Simplifier, and as the basic rule context as “\textit{intro?}”. The rule is called \( t.\text{equality} \).

5. Representations of arbitrary record expressions as canonical constructor terms are provided both in \textit{cases} and \textit{induct} format (cf. the generic proof methods of the same name, \$6.6\). Several variations are available, for fixed records, record schemes, more parts etc.

The generic proof methods are sufficiently smart to pick the most sensible rule according to the type of the indicated record expression: users just need to apply something like “\textit{(cases r)}” to a certain proof problem.

6. The derived record operations \( t.\text{make} \), \( t.\text{fields} \), \( t.\text{extend} \), \( t.\text{truncate} \) are \textit{not} treated automatically, but usually need to be expanded by hand, using the collective fact \( t.\text{defs} \).
Examples
See "~/src/HOL/ex/Records.thy, for example.

10.6 Adhoc tuples

\textit{split\_format} : attribute

\texttt{split\_format (complete)} causes arguments in function applications to be represented canonically according to their tuple type structure.

Note that this operation tends to invent funny names for new local parameters introduced.

10.7 Typedef axiomatization

\texttt{typedef} : \textit{local\_theory \rightarrow proof(prove)}

A Gordon/HOL-style type definition is a certain axiom scheme that identifies a new type with a subset of an existing type. More precisely, the new type is defined by exhibiting an existing type $\tau$, a set $A :: \tau$ set, and a theorem that proves $\exists x. x \in A$. Thus $A$ is a non-empty subset of $\tau$, and the new type denotes this subset. New functions are postulated that establish an isomorphism between the new type and the subset. In general, the type $\tau$ may involve type variables $\alpha_1, \ldots, \alpha_n$ which means that the type definition produces a type constructor ($\alpha_1, \ldots, \alpha_n$) $t$ depending on those type arguments.

The axiomatization can be considered a “definition” in the sense of the particular set-theoretic interpretation of HOL [38], where the universe of types is required to be downwards-closed wrt. arbitrary non-empty subsets. Thus genuinely new types introduced by \texttt{typedef} stay within the range of HOL models by construction. Note that \texttt{type\_synonym} from Isabelle/Pure merely introduces syntactic abbreviations, without any logical significance.
**typedef** $(\alpha_1, \ldots, \alpha_n) \ t = A$ axiomatizes a type definition in the background theory of the current context, depending on a non-emptiness result of the set $A$ that needs to be proven here. The set $A$ may contain type variables $\alpha_1, \ldots, \alpha_n$ as specified on the LHS, but no term variables.

Even though a local theory specification, the newly introduced type constructor cannot depend on parameters or assumptions of the context: this is structurally impossible in HOL. In contrast, the non-emptiness proof may use local assumptions in unusual situations, which could result in different interpretations in target contexts: the meaning of the bijection between the representing set $A$ and the new type $t$ may then change in different application contexts.

By default, **typedef** defines both a type constructor $t$ for the new type, and a term constant $t$ for the representing set within the old type. Use the “*(open)*” option to suppress a separate constant definition altogether. The injection from type to set is called $\text{Rep}_t$, its inverse $\text{Abs}_t$, unless explicit **morphism** specification provides alternative names.
The core axiomatization uses the locale predicate \textit{type\_definition} as defined in Isabelle/HOL. Various basic consequences of that are instantiated accordingly, re-using the locale facts with names derived from the new type constructor. Thus the generic \textit{type\_definition}.Rep is turned into the specific \textit{Rep}\_t, for example.

Theorems \textit{type\_definition}.Rep, \textit{type\_definition}.Rep\_inverse, and \textit{type\_definition}.Abs\_inverse provide the most basic characterization as a corresponding injection/surjection pair (in both directions). The derived rules \textit{type\_definition}.Rep\_inject and \textit{type\_definition}.Abs\_inject provide a more convenient version of injectivity, suitable for automated proof tools (e.g. in declarations involving \textit{simp} or \textit{iff}). Furthermore, the rules \textit{type\_definition}.Rep\_cases / \textit{type\_definition}.Rep\_induct, and \textit{type\_definition}.Abs\_cases / \textit{type\_definition}.Abs\_induct provide alternative views on surjectivity. These rules are already declared as set or type rules for the generic \textit{cases} and \textit{induct} methods, respectively.

An alternative name for the set definition (and other derived entities) may be specified in parentheses; the default is to use \textit{t} directly.

If you introduce a new type axiomatically, i.e. via \texttt{typedef} and \texttt{axiomatization}, the minimum requirement is that it has a non-empty model, to avoid immediate collapse of the HOL logic. Moreover, one needs to demonstrate that the interpretation of such free-form axiomatizations can coexist with that of the regular \texttt{typedef} scheme, and any extension that other people might have introduced elsewhere (e.g. in HOLCF [21]).

\textbf{Examples}

Type definitions permit the introduction of abstract data types in a safe way, namely by providing models based on already existing types. Given some abstract axiomatic description \(P\) of a type, this involves two steps:

1. Find an appropriate type \(\tau\) and subset \(A\) which has the desired properties \(P\), and make a type definition based on this representation.

2. Prove that \(P\) holds for \(\tau\) by lifting \(P\) from the representation.

You can later forget about the representation and work solely in terms of the abstract properties \(P\).

The following trivial example pulls a three-element type into existence within the formal logical environment of HOL.
typedef three = \{(True, True), (True, False), (False, True)\}
by blast

definition One = Abs_three (True, True)
definition Two = Abs_three (True, False)
definition Three = Abs_three (False, True)

lemma three_distinct: One \neq Two One \neq Three Two \neq Three
by (simp_all add: One_def Two_def Three_def Abs_three_inject three_def)

lemma three_cases:
  fixes x :: three obtains x = One | x = Two | x = Three
by (cases x) (auto simp: One_def Two_def Three_def Abs_three_inject three_def)

Note that such trivial constructions are better done with derived specification mechanisms such as datatype:

datatype three' = One' | Two' | Three'

This avoids re-doing basic definitions and proofs from the primitive typedef above.

10.8 Functorial structure of types

enriched_type : local_theory \rightarrow proof(prove)

enriched_type prefix: m allows to prove and register properties about the functorial structure of type constructors. These properties then can be used by other packages to deal with those type constructors in certain type constructions. Characteristic theorems are noted in the current local theory. By default, they are prefixed with the base name of the type constructor, an explicit prefix can be given alternatively.

The given term m is considered as mapper for the corresponding type constructor and must conform to the following type pattern:

\[ m :: \sigma_1 \Rightarrow \ldots \sigma_k \Rightarrow (\alpha_n) t \Rightarrow (\beta_n) t \]
where $t$ is the type constructor, $\alpha_1$ and $\beta_1$ are distinct type variables free in the local theory and $\sigma_1, \ldots, \sigma_k$ is a subsequence of $\alpha_1 \Rightarrow \beta_1$, $\beta_1 \Rightarrow \alpha_1, \ldots, \alpha_n \Rightarrow \beta_n, \beta_n \Rightarrow \alpha_n$.

10.9 Transfer package

- transfer : method
- transfer' : method
- transfer_prover : method
- transfer_rule : attribute
- relator_eq : attribute

*transfer* method replaces the current subgoal with a logically equivalent one that uses different types and constants. The replacement of types and constants is guided by the database of transfer rules. Goals are generalized over all free variables by default; this is necessary for variables whose types change, but can be overridden for specific variables with e.g. *transfer fixing*: $x \ y \ z$.

*transfer‘* is a variant of *transfer* that allows replacing a subgoal with one that is logically stronger (rather than equivalent). For example, a subgoal involving equality on a quotient type could be replaced with a subgoal involving equality (instead of the corresponding equivalence relation) on the underlying raw type.

*transfer_prover* method assists with proving a transfer rule for a new constant, provided the constant is defined in terms of other constants that already have transfer rules. It should be applied after unfolding the constant definitions.

*transfer_rule* attribute maintains a collection of transfer rules, which relate constants at two different types. Typical transfer rules may relate different type instances of the same polymorphic constant, or they may relate an operation on a raw type to a corresponding operation on an abstract type (quotient or subtype). For example:

\[
((A \Rightarrow \Rightarrow B) \Rightarrow \Rightarrow list_all2 \ A \Rightarrow \Rightarrow list_all2 \ B) \ map \ map \\
(cr_int \Rightarrow \Rightarrow cr_int \Rightarrow \Rightarrow cr_int) \ (\lambda(x,y) (u,v) (x+u, y+v)) \ plus
\]

Lemmas involving predicates on relations can also be registered using the same attribute. For example:
\textit{bi\_unique} A \implies (\textit{list\_all2} A \implies \textit{op} =) \textit{distinct} \textit{distinct} \\
\lbrack \textit{bi\_unique} A; \textit{bi\_unique} B \rbrack \implies \textit{bi\_unique} (\textit{prod\_rel} A B)

\textit{relator\_eq} attribute collects identity laws for relators of various type constructors, e.g. \textit{list\_all2} (\textit{op} =) = (\textit{op} =). The \textit{transfer} method uses these lemmas to infer transfer rules for non-polymorphic constants on the fly.

10.10 Lifting package

\begin{itemize}
\item \textbf{setup\_lifting} : \textit{local\_theory} \rightarrow \textit{local\_theory}
\item \textbf{lift\_definition} : \textit{local\_theory} \rightarrow \textit{proof} (\textit{prove})
\item \textbf{print\_quotmaps} : \textit{context} \rightarrow
\item \textbf{print\_quotients} : \textit{context} \rightarrow
\item \textbf{quot\_map} : \textit{attribute}
\item \textbf{invariant\_commute} : \textit{attribute}
\end{itemize}

\textbf{setup\_lifting} Sets up the Lifting package to work with a user-defined type. The user must provide either a quotient theorem \textit{Quotient R Abs Rep}
$T$ or a type definition theorem $\text{type.definition Rep Abs A}$. The package configures transfer rules for equality and quantifiers on the type, and sets up the $\text{lift.definition}$ command to work with the type. In the case of a quotient theorem, an optional theorem $\text{reflp R}$ can be provided as a second argument. This allows the package to generate stronger transfer rules.

$\text{setup_lifting}$ is called automatically if a quotient type is defined by the command $\text{quotient_type}$ from the Quotient package.

If $\text{setup_lifting}$ is called with a type definition theorem, the abstract type implicitly defined by the theorem is declared as an abstract type in the code generator. This allows $\text{lift.definition}$ to register (generated) code certificate theorems as abstract code equations in the code generator. The option $\text{no_abs_code}$ of the command $\text{setup_lifting}$ can turn off that behavior and causes that code certificate theorems generated by $\text{lift.definition}$ are not registered as abstract code equations.

$\text{lift.definition} \ f :: \tau \thinspace \text{is} \thinspace \ t$ Defines a new function $f$ with an abstract type $\tau$ in terms of a corresponding operation $t$ on a representation type. The term $t$ doesn’t have to be necessarily a constant but it can be any term. Users must discharge a respectfulness proof obligation when each constant is defined. For a type copy, i.e. a typedef with $\text{UNIV}$, the proof is discharged automatically. The obligation is presented in a user-friendly, readable form. A respectfulness theorem in the standard format $f.rsp$ and a transfer rule $f.transfer$ for the Transfer package are generated by the package.

Integration with code_abstype: For typedefs (e.g. subtypes corresponding to a datatype invariant, such as dlist), $\text{lift.definition}$ generates a code certificate theorem $f.rep_eq$ and sets up code generation for each constant.

$\text{print_quotmaps}$ prints stored quotient map theorems.

$\text{print_quotients}$ prints stored quotient theorems.

$\text{quot_map}$ registers a quotient map theorem. For examples see $\sim$/src/HOL/Library/Quotient_List.thy or other Quotient_*.thy files.

$\text{invariant_commute}$ registers a theorem which shows a relationship between the constant $\text{Lifting.invariant}$ (used for internal encoding of proper subtypes) and a relator. Such theorems allows the package to hide
Lifting invariant from a user in a user-readable form of a respectfulness theorem. For examples see ~/src/HOL/Library/Quotient_List.thy or other Quotient_*.thy files.

10.11 Quotient types

\[
\text{quotient_type} : \text{local\_theory} \to \text{proof(prove)}
\]
\[
\text{quotient\_definition} : \text{local\_theory} \to \text{proof(prove)}
\]
\[
\text{print\_quotmapsQ3} : \text{context} \to
\]
\[
\text{print\_quotientsQ3} : \text{context} \to
\]
\[
\text{print\_quotconsts} : \text{context} \to
\]
\[
\text{lifting} : \text{method}
\]
\[
\text{lifting\_setup} : \text{method}
\]
\[
\text{descending} : \text{method}
\]
\[
\text{descending\_setup} : \text{method}
\]
\[
\text{partiality\_descending} : \text{method}
\]
\[
\text{partiality\_descending\_setup} : \text{method}
\]
\[
\text{regularize} : \text{method}
\]
\[
\text{injection} : \text{method}
\]
\[
\text{cleaning} : \text{method}
\]
\[
\text{quot\_thm} : \text{attribute}
\]
\[
\text{quot\_lifted} : \text{attribute}
\]
\[
\text{quot\_respect} : \text{attribute}
\]
\[
\text{quot\_preserve} : \text{attribute}
\]

The quotient package defines a new quotient type given a raw type and a partial equivalence relation. It also includes automation for transporting definitions and theorems. It can automatically produce definitions and theorems on the quotient type, given the corresponding constants and facts on the raw type.
CHAPTER 10. ISABELLE/HOL

\section*{spec}

\begin{itemize}
\item \texttt{typespec}
\item \texttt{mixfix}
\item \texttt{type} \(/\) \texttt{term}
\item \texttt{partial} \(:\)
\item \texttt{morphism}s \texttt{name} \texttt{name}
\item \texttt{quotient\_definition}
\item \texttt{constdecl}
\item \texttt{thmdecl}
\item \texttt{term} \texttt{is} \texttt{term}
\end{itemize}

\section*{constdecl}

\begin{itemize}
\item \texttt{name}
\item \texttt{::} \texttt{type} \texttt{mixfix}
\end{itemize}

\section*{lifting}

\begin{itemize}
\item \texttt{thmrefs}
\end{itemize}

\section*{lifting_setup}

\begin{itemize}
\item \texttt{thmrefs}
\end{itemize}
quotient_type defines quotient types. The injection from a quotient type to a raw type is called \textit{rep}, its inverse \textit{abs} unless explicit \textbf{morphisms} specification provides alternative names. quotient_type requires the user to prove that the relation is an equivalence relation (predicate \textit{equivp}), unless the user specifies explicitly \textit{partial} in which case the obligation is \textit{part_equiv}. A quotient defined with \textit{partial} is weaker in the sense that less things can be proved automatically.

\textbf{quotient_definition} defines a constant on the quotient type.

\textbf{print_quotmapsQ3} prints quotient map functions.

\textbf{print_quotientsQ3} prints quotients.

\textbf{print_quotconsts} prints quotient constants.

\textit{lifting} and \textit{lifting_setup} methods match the current goal with the given raw theorem to be lifted producing three new subgoals: regularization, injection and cleaning subgoals. \textit{lifting} tries to apply the heuristics for automatically solving these three subgoals and leaves only the subgoals unsolved by the heuristics to the user as opposed to \textit{lifting_setup} which leaves the three subgoals unsolved.

\textit{descending} and \textit{descending_setup} try to guess a raw statement that would lift to the current subgoal. Such statement is assumed as a new subgoal and \textit{descending} continues in the same way as \textit{lifting} does. \textit{descending} tries to solve the arising regularization, injection and cleaning subgoals with the analogous method \textit{descending_setup} which leaves the four unsolved subgoals.

\textit{partiality_descending} finds the regularized theorem that would lift to the current subgoal, lifts it and leaves as a subgoal. This method can be used with partial equivalence quotients where the non regularized statements would not be true. \textit{partiality_descending_setup} leaves the injection and cleaning subgoals unchanged.

\textit{regularize} applies the regularization heuristics to the current subgoal.

\textit{injection} applies the injection heuristics to the current goal using the stored quotient respectfulness theorems.

\textit{cleaning} applies the injection cleaning heuristics to the current subgoal using the stored quotient preservation theorems.
quot_lifted attribute tries to automatically transport the theorem to the quotient type. The attribute uses all the defined quotients types and quotient constants often producing undesired results or theorems that cannot be lifted.

quot_respect and quot_preserve attributes declare a theorem as a respectfulness and preservation theorem respectively. These are stored in the local theory store and used by the injection and cleaning methods respectively.

quot_thm declares that a certain theorem is a quotient extension theorem. Quotient extension theorems allow for quotienting inside container types. Given a polymorphic type that serves as a container, a map function defined for this container using enriched type and a relation map defined for for the container type, the quotient extension theorem should be Quotient3 $R\ Abs\ Rep \implies Quotient3\ (rel\_map\ R)\ (map\ Abs)\ (map\ Rep)$. Quotient extension theorems are stored in a database and are used all the steps of lifting theorems.

10.12 Coercive subtyping

coercion : attribute
coercion_enabled : attribute
coercion_map : attribute

Coercive subtyping allows the user to omit explicit type conversions, also called coercions. Type inference will add them as necessary when parsing a term. See [41] for details.
coercion \( f \) registers a new coercion function \( f :: \sigma_1 \Rightarrow \sigma_2 \) where \( \sigma_1 \) and \( \sigma_2 \) are type constructors without arguments. Coercions are composed by the inference algorithm if needed. Note that the type inference algorithm is complete only if the registered coercions form a lattice.

coercion_map \( map \) registers a new map function to lift coercions through type constructors. The function \( map \) must conform to the following type pattern

\[
\text{map} :: f_1 \Rightarrow \ldots \Rightarrow f_n \Rightarrow (\alpha_1, \ldots, \alpha_n) \ t \Rightarrow (\beta_1, \ldots, \beta_n) \ t
\]

where \( t \) is a type constructor and \( f_i \) is of type \( \alpha_i \Rightarrow \beta_i \) or \( \beta_i \Rightarrow \alpha_i \). Registering a map function overwrites any existing map function for this particular type constructor.

coercion_enabled enables the coercion inference algorithm.

### 10.13 Arithmetic proof support

\texttt{arith} : method

\texttt{arith} : attribute

\texttt{arith_split} : attribute

\texttt{arith} decides linear arithmetic problems (on types \texttt{nat}, \texttt{int}, \texttt{real}). Any current facts are inserted into the goal before running the procedure.

\texttt{arith} declares facts that are supplied to the arithmetic provers implicitly.

\texttt{arith_split} attribute declares case split rules to be expanded before \texttt{arith} is invoked.

Note that a simpler (but faster) arithmetic prover is already invoked by the Simplifier.

### 10.14 Intuitionistic proof search

\texttt{iProver} : method
**iprover** performs intuitionistic proof search, depending on specifically declared rules from the context, or given as explicit arguments. Chained facts are inserted into the goal before commencing proof search.

Rules need to be classified as *intro*, *elim*, or *dest*; here the “!” indicator refers to “safe” rules, which may be applied aggressively (without considering back-tracking later). Rules declared with “?” are ignored in proof search (the single-step *rule* method still observes these). An explicit weight annotation may be given as well; otherwise the number of rule premises will be taken into account here.

### 10.15 Model Elimination and Resolution

**meson** : method

**metis** : method

*meson* implements Loveland’s model elimination procedure [18]. See ~/src/HOL/ex/Meson_Test.thy for examples.
metis combines ordered resolution and ordered paramodulation to find first-order (or mildly higher-order) proofs. The first optional argument specifies a type encoding; see the Sledgehammer manual [7] for details. The directory "/src/HOL/Metis_Examples contains several small theories developed to a large extent using metis.

10.16 Coherent Logic

coherent : method

coherent solves problems of Coherent Logic [6], which covers applications in confluence theory, lattice theory and projective geometry. See ~src/HOL/ex/Coherent.thy for some examples.

10.17 Proving propositions

In addition to the standard proof methods, a number of diagnosis tools search for proofs and provide an Isar proof snippet on success. These tools are available via the following commands.

solve_direct* : proof →
try* : proof →
try0* : proof →
sledgehammer* : proof →
sledgehammer_params : theory → theory
solve\_direct  checks whether the current subgoals can be solved directly by an existing theorem. Duplicate lemmas can be detected in this way.
try0 attempts to prove a subgoal using a combination of standard proof methods (auto, simp, blast, etc.). Additional facts supplied via simp:, intro:, elim:, and dest: are passed to the appropriate proof methods.

try attempts to prove or disprove a subgoal using a combination of provers and disprovers (solve_direct, quickcheck, try0, sledgehammer, nitpick).


sledgehammer_params changes sledgehammer configuration options persistently.

10.18 Checking and refuting propositions

Identifying incorrect propositions usually involves evaluation of particular assignments and systematic counterexample search. This is supported by the following commands.

\[
\begin{align*}
\text{value}^* & : \text{context} \to \text{term} \\
\text{values}^* & : \text{context} \to \text{term} \\
\text{quickcheck}^* & : \text{proof} \to \text{term} \\
\text{refute}^* & : \text{proof} \to \text{term} \\
\text{nitpick}^* & : \text{proof} \to \text{term} \\
\text{quickcheck_params} & : \text{theory} \to \text{theory} \\
\text{refute_params} & : \text{theory} \to \text{theory} \\
\text{nitpick_params} & : \text{theory} \to \text{theory} \\
\text{quickcheck_generator} & : \text{theory} \to \text{theory} \\
\text{find_unused_assms} & : \text{context} \to \text{term}
\end{align*}
\]
value $t$ evaluates and prints a term; optionally modes can be specified, which are appended to the current print mode; see §7.1.3. Internally, the evaluation is performed by registered evaluators, which are invoked
sequentially until a result is returned. Alternatively a specific evaluator can be selected using square brackets; typical evaluators use the current set of code equations to normalize and include simp for fully symbolic evaluation using the simplifier, nbe for normalization by evaluation and code for code generation in SML.

**values** \( t \) enumerates a set comprehension by evaluation and prints its values up to the given number of solutions; optionally **modes** can be specified, which are appended to the current print mode; see §7.1.3.

**quickcheck** tests the current goal for counterexamples using a series of assignments for its free variables; by default the first subgoal is tested, an other can be selected explicitly using an optional goal index. Assignments can be chosen exhausting the search space up to a given size, or using a fixed number of random assignments in the search space, or exploring the search space symbolically using narrowing. By default, quickcheck uses exhaustive testing. A number of configuration options are supported for **quickcheck**, notably:

- **tester** specifies which testing approach to apply. There are three testers, **exhaustive**, **random**, and **narrowing**. An unknown configuration option is treated as an argument to tester, making **tester** = **optional**. When multiple testers are given, these are applied in parallel. If no tester is specified, quickcheck uses the testers that are set active, i.e., configurations **quickcheck_exhaustive_active**, **quickcheck_random_active**, **quickcheck_narrowing_active** are set to true.

- **size** specifies the maximum size of the search space for assignment values.

- **genuine_only** sets quickcheck only to return genuine counterexample, but not potentially spurious counterexamples due to underspecified functions.

- **abort_potential** sets quickcheck to abort once it found a potentially spurious counterexample and to not continue to search for a further genuine counterexample. For this option to be effective, the **genuine_only** option must be set to false.

- **eval** takes a term or a list of terms and evaluates these terms under the variable assignment found by quickcheck.

- **iterations** sets how many sets of assignments are generated for each particular size.
no_assms specifies whether assumptions in structured proofs should be ignored.

locale specifies how to process conjectures in a locale context, i.e., they can be interpreted or expanded. The option is a whitespace-separated list of the two words interpret and expand. The list determines the order they are employed. The default setting is to first use interpretations and then test the expanded conjecture. The option is only provided as attribute declaration, but not as parameter to the command.

timeout sets the time limit in seconds.

default_type sets the type(s) generally used to instantiate type variables.

report if set quickcheck reports how many tests fulfilled the preconditions.

use_subtype if set quickcheck automatically lifts conjectures to registered subtypes if possible, and tests the lifted conjecture.

quiet if set quickcheck does not output anything while testing.

verbose if set quickcheck informs about the current size and cardinality while testing.

expect can be used to check if the user’s expectation was met (no_expectation, no_counterexample, or counterexample).

These option can be given within square brackets.

quickcheck_params changes quickcheck configuration options persistently.

quickcheck_generator creates random and exhaustive value generators for a given type and operations. It generates values by using the operations as if they were constructors of that type.

refute tests the current goal for counterexamples using a reduction to SAT. The following configuration options are supported:

minsize specifies the minimum size (cardinality) of the models to search for.

maxsize specifies the maximum size (cardinality) of the models to search for. Nonpositive values mean $\infty$. 
maxvars specifies the maximum number of Boolean variables to use when transforming the term into a propositional formula. Non-positive values mean $\infty$.

satsolver specifies the SAT solver to use.

no_assms specifies whether assumptions in structured proofs should be ignored.

maxtime sets the time limit in seconds.

expect can be used to check if the user’s expectation was met (genuine, potential, none, or unknown).

These option can be given within square brackets.

refute_params changes refute configuration options persistently.

nitpick tests the current goal for counterexamples using a reduction to first-order relational logic. See the Nitpick manual [8] for details.

nitpick_params changes nitpick configuration options persistently.

find_unused_assms finds potentially superfluous assumptions in theorems using quickcheck. It takes the theory name to be checked for superfluous assumptions as optional argument. If not provided, it checks the current theory. Options to the internal quickcheck invocations can be changed with common configuration declarations.

10.19 Unstructured case analysis and induction

The following tools of Isabelle/HOL support cases analysis and induction in unstructured tactic scripts; see also §6.6 for proper Isar versions of similar ideas.

\[
\begin{align*}
\text{case_tac}^* & : method \\
\text{induct_tac}^* & : method \\
\text{ind_cases}^* & : method \\
\text{inductive_cases}^* & : \text{local_theory} \rightarrow \text{local_theory}
\end{align*}
\]
case_tac and induct_tac admit to reason about inductive types. Rules are selected according to the declarations by the cases and induct attributes, cf. §6.6. The datatype package already takes care of this.

These unstructured tactics feature both goal addressing and dynamic instantiation. Note that named rule cases are not provided as would be by the proper cases and induct proof methods (see §6.6). Unlike the induct method, induct_tac does not handle structured rule statements, only the compact object-logic conclusion of the subgoal being addressed.

ind_cases and inductive_cases provide an interface to the internal mk_cases operation. Rules are simplified in an unrestricted forward manner.
While \textit{ind_cases} is a proof method to apply the result immediately as elimination rules, \textbf{inductive_cases} provides case split theorems at the theory level for later use. The \texttt{for} argument of the \textit{ind_cases} method allows to specify a list of variables that should be generalized before applying the resulting rule.

\section*{10.20 Executable code}

For validation purposes, it is often useful to \textit{execute} specifications. In principle, execution could be simulated by Isabelle’s inference kernel, i.e. by a combination of resolution and simplification. Unfortunately, this approach is rather inefficient. A more efficient way of executing specifications is to translate them into a functional programming language such as ML.

Isabelle provides a generic framework to support code generation from executable specifications. Isabelle/HOL instantiates these mechanisms in a way that is amenable to end-user applications. Code can be generated for functional programs (including overloading using type classes) targeting SML [20], OCaml [17], Haskell [37] and Scala [28]. Conceptually, code generation is split up in three steps: \textit{selection} of code theorems, \textit{translation} into an abstract executable view and \textit{serialization} to a specific \textit{target language}. Inductive specifications can be executed using the predicate compiler which

\begin{verbatim}
export_code*: context →
  code: attribute

code_abort: theory → theory

code_datatype: theory → theory

print_codesetup*: context →
  code_unfold: attribute
  code_post: attribute

print_codeproc*: context →
  code_thms*: context →
  code_deps*: context →

code_const: theory → theory

code_type: theory → theory

code_class: theory → theory

code_instance: theory → theory

code_reserved: theory → theory

code_monad: theory → theory

code_include: theory → theory

code_modulename: theory → theory

code_reflect: theory → theory

code_pred: theory → proof(prove)
\end{verbatim}
target

SML
  OCaml
  Haskell
  Scala

code
  del
    abstype
      abstract
  code_abort
    const
  code_datatype
    const
  code_unfold
    del
  code_post
    del
  code_abbrev
syntax

\texttt{string}\n
\texttt{infix} \quad \texttt{nat} \quad \texttt{string}\n
\texttt{infixl} \quad \texttt{infixr}\n
\texttt{mode} \quad \texttt{as} \quad \texttt{const}\n
\textbf{export_code} generates code for a given list of constants in the specified
target language(s). If no serialization instruction is given, only abstract code is generated internally.

Constants may be specified by giving them literally, referring to all executable contents within a certain theory by giving name.*, or referring to all executable constants currently available by giving *.

By default, for each involved theory one corresponding name space module is generated. Alternativly, a module name may be specified after the module_name keyword; then all code is placed in this module.

For SML, OCaml and Scala the file specification refers to a single file; for Haskell, it refers to a whole directory, where code is generated in multiple files reflecting the module hierarchy. Omitting the file specification denotes standard output.

Serializers take an optional list of arguments in parentheses. For SML and OCaml, "no_signatures" omits explicit module signatures.

For Haskell a module name prefix may be given using the "root:" argument; "string_classes" adds a "deriving (Read, Show)" clause to each appropriate datatype declaration.

code explicitly selects (or with option “del” deselects) a code equation for code generation. Usually packages introducing code equations provide a reasonable default setup for selection. Variants code abstype and code abstract declare abstract datatype certificates or code equations on abstract datatype representations respectively.

code_abort declares constants which are not required to have a definition by means of code equations; if needed these are implemented by program abort instead.

code_datatype specifies a constructor set for a logical type.

print_codesetup gives an overview on selected code equations and code generator datatypes.

code_unfold declares (or with option “del” removes) theorems which during preprocessing are applied as rewrite rules to any code equation or evaluation input.

code_post declares (or with option “del” removes) theorems which are applied as rewrite rules to any result of an evaluation.
code_abbrev declares equations which are applied as rewrite rules to any result of an evaluation and symmetrically during preprocessing to any code equation or evaluation input.

print_codeproc prints the setup of the code generator preprocessor.

code_thms prints a list of theorems representing the corresponding program containing all given constants after preprocessing.

code_deps visualizes dependencies of theorems representing the corresponding program containing all given constants after preprocessing.

code_const associates a list of constants with target-specific serializations; omitting a serialization deletes an existing serialization.

code_type associates a list of type constructors with target-specific serializations; omitting a serialization deletes an existing serialization.

code_class associates a list of classes with target-specific class names; omitting a serialization deletes an existing serialization. This applies only to Haskell.

code_instance declares a list of type constructor / class instance relations as “already present” for a given target. Omitting a “−” deletes an existing “already present” declaration. This applies only to Haskell.

code_reserved declares a list of names as reserved for a given target, preventing it to be shadowed by any generated code.

code_monad provides an auxiliary mechanism to generate monadic code for Haskell.

code_include adds arbitrary named content ("include") to generated code. A “−” as last argument will remove an already added “include”.

code_modulename declares aliasings from one module name onto another.

code_reflect without a “file” argument compiles code into the system runtime environment and modifies the code generator setup that future invocations of system runtime code generation referring to one of the “datatypes” or “functions” entities use these precompiled entities. With a “file” argument, the corresponding code is generated into that specified file without modifying the code generator setup.
**code_pred** creates code equations for a predicate given a set of introduction rules. Optional mode annotations determine which arguments are supposed to be input or output. If alternative introduction rules are declared, one must prove a corresponding elimination rule.

### 10.21 Definition by specification

- **specification**: `theory → proof(prove)`
- **ax_specification**: `theory → proof(prove)`

**decl**

- **name**: `term (overloaded)`
- **prop**: `thmdecl`

**specification** `decls ϕ` sets up a goal stating the existence of terms with the properties specified to hold for the constants given in `decls`. After finishing the proof, the theory will be augmented with definitions for the given constants, as well as with theorems stating the properties for these constants.

**ax_specification** `decls ϕ` sets up a goal stating the existence of terms with the properties specified to hold for the constants given in `decls`. After finishing the proof, the theory will be augmented with axioms expressing the properties given in the first place.
decl declares a constant to be defined by the specification given. The definition for the constant \( c \) is bound to the name \( c\_def \) unless a theorem name is given in the declaration. Overloaded constants should be declared as such.

Whether to use specification or ax_specification is to some extent a matter of style. specification introduces no new axioms, and so by construction cannot introduce inconsistencies, whereas ax_specification does introduce axioms, but only after the user has explicitly proven it to be safe. A practical issue must be considered, though: After introducing two constants with the same properties using specification, one can prove that the two constants are, in fact, equal. If this might be a problem, one should use ax_specification.
Isabelle/HOLCF

11.1 Mixfix syntax for continuous operations

\textbf{consts} : \textit{theory} \rightarrow \textit{theory}

HOLCF provides a separate type for continuous functions $\alpha \rightarrow \beta$, with an explicit application operator $f \cdot x$. Isabelle mixfix syntax normally refers directly to the pure meta-level function type $\alpha \Rightarrow \beta$, with application $f x$.

The HOLCF variant of \textbf{consts} modifies that of Pure Isabelle (cf. §5.10.3) such that declarations involving continuous function types are treated specifically. Any given syntax template is transformed internally, generating translation rules for the abstract and concrete representation of continuous application. Note that mixing of HOLCF and Pure application is not supported!

11.2 Recursive domains

\textbf{domain} : \textit{theory} \rightarrow \textit{theory}

\begin{center}
\begin{tikzpicture}
  \node (domain) {domain};
  \node (spec) [right of=domain] {spec};
  \node (parname) [below of=spec] {parname};
  \node (and) [below of=spec] {and};
  \node (typespec) [below of=spec] {typespec};
  \node (cons) [below of=typespec] {cons};
  \node (name) [below of=cons] {name};
  \node (type) [below of=name] {type};
  \node (mixfix) [below of=name] {mixfix};
  \draw[-stealth] (domain) -- (spec);
  \draw[-stealth] (spec) -- (parname);
  \draw[-stealth] (spec) -- (and);
  \draw[-stealth] (typespec) -- (cons);
  \draw[-stealth] (cons) -- (name);
  \draw[-stealth] (name) -- (type);
  \draw[-stealth] (name) -- (mixfix);
\end{tikzpicture}
\end{center}

\textit{spec}

\begin{center}
\begin{tikzpicture}
  \node (typespec) {typespec};
  \node (cons) [below of=typespec] {cons};
  \node (1) [below of=cons] {1};
  \draw[-stealth] (typespec) -- (cons);
  \draw[-stealth] (cons) -- (1);
\end{tikzpicture}
\end{center}

\textit{cons}

\begin{center}
\begin{tikzpicture}
  \node (name) {name};
  \node (type) [below of=name] {type};
  \node (mixfix) [below of=name] {mixfix};
  \draw[-stealth] (name) -- (type);
  \draw[-stealth] (name) -- (mixfix);
\end{tikzpicture}
\end{center}
Recursive domains in HOLCF are analogous to datatypes in classical HOL (cf. §10.4). Mutual recursion is supported, but no nesting nor arbitrary branching. Domain constructors may be strict (default) or lazy, the latter admits to introduce infinitary objects in the typical LCF manner (e.g. lazy lists). See also [21] for a general discussion of HOLCF domains.
12.1 Type checking

The ZF logic is essentially untyped, so the concept of “type checking” is performed as logical reasoning about set-membership statements. A special method assists users in this task; a version of this is already declared as a “solver” in the standard Simplifier setup.

\[
\text{print\_tcset}^* : \text{context} \rightarrow \\
\text{typecheck} : \text{method} \\
\text{TC} : \text{attribute}
\]

\[\text{print\_tcset}\] prints the collection of typechecking rules of the current context.

\[\text{typecheck}\] attempts to solve any pending type-checking problems in subgoals.

\[\text{TC}\] adds or deletes type-checking rules from the context.

12.2 (Co)Inductive sets and datatypes

12.2.1 Set definitions

In ZF everything is a set. The generic inductive package also provides a specific view for “datatype” specifications. Coinductive definitions are available
in both cases, too.

\[
\begin{align*}
\text{inductive} & : \text{theory} \rightarrow \text{theory} \\
\text{coinductive} & : \text{theory} \rightarrow \text{theory} \\
\text{datatype} & : \text{theory} \rightarrow \text{theory} \\
\text{codatatype} & : \text{theory} \rightarrow \text{theory}
\end{align*}
\]
In the following syntax specification `monos`, `typeintros`, and `typeelims` are the same as above.

```
datatype codatatype domain
```

```
domain
<=
```

```
dtspec
```

```
con
```

```
ame
```
See [30] for further information on inductive definitions in ZF, but note that this covers the old-style theory format.

### 12.2.2 Primitive recursive functions

**primrec** : \( \text{theory} \rightarrow \text{theory} \)

### 12.2.3 Cases and induction: emulating tactic scripts

The following important tactical tools of Isabelle/ZF have been ported to Isar. These should not be used in proper proof texts.

- \( \text{case_tac}^* \) : \text{method}
- \( \text{induct_tac}^* \) : \text{method}
- \( \text{ind_cases}^* \) : \text{method}
- \( \text{inductive_cases} \) : \( \text{theory} \rightarrow \text{theory} \)
inductive_cases

prop

thmdecl

and
Part IV
Appendix
Appendix A

Isabelle/Isar quick reference

A.1 Proof commands

A.1.1 Primitives and basic syntax

- **fix** \( x \)  
  augment context by \( \forall x. \square \)

- **assume** \( a : \varphi \)  
  augment context by \( \varphi \Rightarrow \square \)

- **then**  
  indicate forward chaining of facts

- **have** \( a : \varphi \)  
  prove local result

- **show** \( a : \varphi \)  
  prove local result, refining some goal

- **using** \( a \)  
  indicate use of additional facts

- **unfolding** \( a \)  
  unfold definitional equations

- **proof** \( m_1 \ldots \) **qed** \( m_2 \)  
  indicate proof structure and refinements

- **\{ ... \}**  
  indicate explicit blocks

- **next**  
  switch blocks

- **note** \( a = b \)  
  reconsider facts

- **let** \( p = t \)  
  abbreviate terms by higher-order matching

- **write** \( c \) (\( mx \))  
  declare local mixfix syntax

\[
\begin{align*}
\text{proof} &= \quad \text{prfx}^* \text{ proof method}^* \; \text{stmt}^* \; \text{qed method}^* \\
\text{prfx} &= \quad \text{apply method} \\
\text{stmt} &= \quad \{ \text{stmt}^* \} \\
\text{goal} &= \quad \text{have name: props proof} \\
\end{align*}
\]
A.1.2 Abbreviations and synonyms

\[\begin{align*}
\text{by } m_1 m_2 & \equiv \text{proof } m_1 \text{ qed } m_2 \\
.. & \equiv \text{by rule} \\
. & \equiv \text{by this} \\
hence & \equiv \text{then have} \\
thus & \equiv \text{then show} \\
\text{from } a & \equiv \text{note } a \text{ then} \\
\text{with } a & \equiv \text{from } a \text{ and this} \\
\text{from this} & \equiv \text{then} \\
\text{from this have} & \equiv \text{hence} \\
\text{from this show} & \equiv \text{thus}
\end{align*}\]

A.1.3 Derived elements

\[\begin{align*}
\text{also}_0 & \approx \text{note calculation } = \text{this} \\
\text{also}_{a+1} & \approx \text{note calculation } = \text{trans } [OF \text{ calculation } \text{this}] \\
\text{finally} & \approx \text{also from calculation} \\
\text{moreover} & \approx \text{note calculation } = \text{calculation } \text{this} \\
\text{ultimately} & \approx \text{moreover from calculation} \\
\text{presume } a: \varphi & \approx \text{assume } a: \varphi \\
\text{def } a: x \equiv t & \approx \text{fix } x \text{ assume } a: x \equiv t \\
\text{obtain } x \text{ where } a: \varphi & \approx \ldots \text{fix } x \text{ assume } a: \varphi \\
\text{case } c & \approx \text{fix } x \text{ assume } c: \varphi \\
\text{sorry} & \approx \text{by cheating}
\end{align*}\]

A.1.4 Diagnostic commands

pr \quad \text{print current state} \\
thm a \quad \text{print fact} \\
prop \varphi \quad \text{print proposition} \\
term t \quad \text{print term} \\
typ \tau \quad \text{print type}
A.2 Proof methods

Single steps (forward-chaining facts)

- assumption: apply some assumption
- this: apply current facts
- rule $a$: apply some rule
- contradiction: apply $\neg$ elimination rule (any order)
- cases $t$: case analysis (provides cases)
- induct $x$: proof by induction (provides cases)

Repeated steps (inserting facts)

- no rules
- intro $a$: introduction rules
- intro_classes: class introduction rules
- elim $a$: elimination rules
- unfold $a$: definitional rewrite rules

Automated proof tools (inserting facts)

- iprover: intuitionistic proof search
- blast, fast: Classical Reasoner
- simp, simp_all: Simplifier (+ Splitter)
- auto, force: Simplifier + Classical Reasoner
- arith: Arithmetic procedures
A.3 Attributes

Rules
- **OF a** rule resolved with facts (skipping “_”)
- **of t** rule instantiated with terms (skipping “_”)
- **where x = t** rule instantiated with terms, by variable name
- **symmetric** resolution with symmetry rule
- **THEN b** resolution with another rule
- **rule_format** result put into standard rule format
- **elim_format** destruct rule turned into elimination rule format

Declarations
- **simp** Simplifier rule
- **intro, elim, dest** Pure or Classical Reasoner rule
- **iff** Simplifier + Classical Reasoner rule
- **split** case split rule
- **trans** transitivity rule
- **sym** symmetry rule

A.4 Rule declarations and methods

<table>
<thead>
<tr>
<th></th>
<th>rule</th>
<th>iprover</th>
<th>blast</th>
<th>simp</th>
<th>auto</th>
<th>fast</th>
<th>simp_all</th>
<th>force</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure.elim! Pure.intro!</td>
<td>×</td>
<td>×</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pure.elim Pure.intro</td>
<td>×</td>
<td>×</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>elim! intro</td>
<td>×</td>
<td>×</td>
<td></td>
<td>×</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>elim intro</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iff</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iff?</td>
<td>×</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>elim? intro?</td>
<td>×</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>simp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>×</td>
</tr>
<tr>
<td>cong</td>
<td></td>
<td>×</td>
<td></td>
<td>×</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>split</td>
<td></td>
<td>×</td>
<td></td>
<td>×</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A.5 Emulating tactic scripts

A.5.1 Commands

- **apply** \( m \)  
  apply proof method at initial position
- **apply_end** \( m \)  
  apply proof method near terminal position
- **done**  
  complete proof
- **defer** \( n \)  
  move subgoal to end
- **prefer** \( n \)  
  move subgoal to beginning
- **back**  
  backtrack last command

A.5.2 Methods

- **rule_tac** insts  
  resolution (with instantiation)
- **erule_tac** insts  
  elim-resolution (with instantiation)
- **drule_tac** insts  
  destruct-resolution (with instantiation)
- **frule_tac** insts  
  forward-resolution (with instantiation)
- **cut_tac** insts  
  insert facts (with instantiation)
- **thin_tac** \( \varphi \)  
  delete assumptions
- **subgoal_tac** \( \varphi \)  
  new claims
- **rename_tac** \( x \)  
  rename innermost goal parameters
- **rotate_tac** \( n \)  
  rotate assumptions of goal
- **tactic** text  
  arbitrary ML tactic
- **case_tac** \( t \)  
  exhaustion (datatypes)
- **induct_tac** \( x \)  
  induction (datatypes)
- **ind_cases** \( t \)  
  exhaustion + simplification (inductive predicates)
Predefined Isabelle symbols

Isabelle supports an infinite number of non-ASCII symbols, which are represented in source text as \textbackslash{name} (where name may be any identifier). It is left to front-end tools how to present these symbols to the user. The collection of predefined standard symbols given below is available by default for Isabelle document output, due to appropriate definitions of isasymlong for each \textbackslash{name} in the isabellesym.sty file. Most of these symbols are displayed properly in Proof General and Isabelle/jEdit. Moreover, any single symbol (or ASCII character) may be prefixed by \textbackslash{sup}, for superscript and \textbackslash{sub}, for subscript, such as A\textbackslash{sup}\textbackslash{star}, for A\textasciistar for A. The alternative versions \textbackslash{isub} and \textbackslash{isup} are considered as quasi letters and may be used within identifiers. Sub- and superscripts that span a region of text are marked up with \textbackslash{bsub}...\textbackslash{esub}, and \textbackslash{bsup}...\textbackslash{esup} respectively. Furthermore, all ASCII characters and most other symbols may be printed in bold by prefixing \textbackslash{bold} such as \textbackslash{bold}\textbackslash{alpha} for α. Note that \textbackslash{bold}, may not be combined with sub- or superscripts for single symbols. Further details of Isabelle document preparation are covered in chapter 4.

\begin{tabular}{ll}
\texttt{\textbackslash{zero}} & 0 \\
\texttt{\textbackslash{two}} & 2 \\
\texttt{\textbackslash{four}} & 4 \\
\texttt{\textbackslash{six}} & 6 \\
\texttt{\textbackslash{eight}} & 8 \\
\texttt{\textbackslash{A}} & A \\
\texttt{\textbackslash{C}} & C \\
\texttt{\textbackslash{E}} & E \\
\texttt{\textbackslash{G}} & G \\
\texttt{\textbackslash{I}} & I \\
\texttt{\textbackslash{K}} & K \\
\texttt{\textbackslash{M}} & M \\
\texttt{\textbackslash{O}} & O \\
\end{tabular}

\begin{tabular}{ll}
\texttt{\textbackslash{one}} & 1 \\
\texttt{\textbackslash{three}} & 3 \\
\texttt{\textbackslash{five}} & 5 \\
\texttt{\textbackslash{seven}} & 7 \\
\texttt{\textbackslash{nine}} & 9 \\
\texttt{\textbackslash{B}} & B \\
\texttt{\textbackslash{D}} & D \\
\texttt{\textbackslash{F}} & F \\
\texttt{\textbackslash{H}} & H \\
\texttt{\textbackslash{J}} & J \\
\texttt{\textbackslash{L}} & L \\
\texttt{\textbackslash{N}} & N \\
\texttt{\textbackslash{P}} & P \\
\end{tabular}
### APPENDIX B. PREDEFINED ISABELLE SYMBOLS

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Symbol</th>
<th>Symbol</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{Q}$</td>
<td>$\mathcal{R}$</td>
<td>$\mathcal{S}$</td>
<td>$\mathcal{T}$</td>
</tr>
<tr>
<td>$\mathcal{U}$</td>
<td>$\mathcal{V}$</td>
<td>$\mathcal{W}$</td>
<td>$\mathcal{X}$</td>
</tr>
<tr>
<td>$\mathcal{Y}$</td>
<td>$\mathcal{Z}$</td>
<td>$a$</td>
<td>$b$</td>
</tr>
<tr>
<td>$c$</td>
<td>$d$</td>
<td>$e$</td>
<td>$f$</td>
</tr>
<tr>
<td>$g$</td>
<td>$h$</td>
<td>$i$</td>
<td>$j$</td>
</tr>
<tr>
<td>$k$</td>
<td>$l$</td>
<td>$m$</td>
<td>$n$</td>
</tr>
<tr>
<td>$o$</td>
<td>$p$</td>
<td>$q$</td>
<td>$r$</td>
</tr>
<tr>
<td>$s$</td>
<td>$t$</td>
<td>$u$</td>
<td>$v$</td>
</tr>
<tr>
<td>$w$</td>
<td>$x$</td>
<td>$y$</td>
<td>$z$</td>
</tr>
<tr>
<td>$\mathcal{A}$</td>
<td>$\mathcal{B}$</td>
<td>$\mathcal{C}$</td>
<td>$\mathcal{D}$</td>
</tr>
<tr>
<td>$\mathcal{E}$</td>
<td>$\mathcal{F}$</td>
<td>$\mathcal{G}$</td>
<td>$\mathcal{H}$</td>
</tr>
<tr>
<td>$\mathcal{I}$</td>
<td>$\mathcal{J}$</td>
<td>$\mathcal{K}$</td>
<td>$\mathcal{L}$</td>
</tr>
<tr>
<td>$\mathcal{M}$</td>
<td>$\mathcal{N}$</td>
<td>$\mathcal{O}$</td>
<td>$\mathcal{P}$</td>
</tr>
<tr>
<td>$\mathcal{Q}$</td>
<td>$\mathcal{R}$</td>
<td>$\mathcal{S}$</td>
<td>$\mathcal{T}$</td>
</tr>
<tr>
<td>$\mathcal{U}$</td>
<td>$\mathcal{V}$</td>
<td>$\mathcal{W}$</td>
<td>$\mathcal{X}$</td>
</tr>
<tr>
<td>$\mathcal{Y}$</td>
<td>$\mathcal{Z}$</td>
<td>$a$</td>
<td>$b$</td>
</tr>
<tr>
<td>$c$</td>
<td>$d$</td>
<td>$e$</td>
<td>$f$</td>
</tr>
<tr>
<td>$g$</td>
<td>$h$</td>
<td>$i$</td>
<td>$j$</td>
</tr>
<tr>
<td>$k$</td>
<td>$l$</td>
<td>$m$</td>
<td>$n$</td>
</tr>
<tr>
<td>$o$</td>
<td>$p$</td>
<td>$q$</td>
<td>$r$</td>
</tr>
<tr>
<td>$s$</td>
<td>$t$</td>
<td>$u$</td>
<td>$v$</td>
</tr>
<tr>
<td>$w$</td>
<td>$x$</td>
<td>$y$</td>
<td>$z$</td>
</tr>
<tr>
<td>\qq</td>
<td>q</td>
<td>\rr</td>
<td>r</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>\ss</td>
<td>s</td>
<td>\tt</td>
<td>t</td>
</tr>
<tr>
<td>\uu</td>
<td>u</td>
<td>\vv</td>
<td>v</td>
</tr>
<tr>
<td>\ww</td>
<td>w</td>
<td>\xx</td>
<td>x</td>
</tr>
<tr>
<td>\yy</td>
<td>y</td>
<td>\zz</td>
<td>z</td>
</tr>
<tr>
<td>\alpha</td>
<td>α</td>
<td>\beta</td>
<td>β</td>
</tr>
<tr>
<td>\gamma</td>
<td>γ</td>
<td>\delta</td>
<td>δ</td>
</tr>
<tr>
<td>\epsilon</td>
<td>ε</td>
<td>\zeta</td>
<td>ζ</td>
</tr>
<tr>
<td>\eta</td>
<td>η</td>
<td>\theta</td>
<td>θ</td>
</tr>
<tr>
<td>\iota</td>
<td>ι</td>
<td>\kappa</td>
<td>κ</td>
</tr>
<tr>
<td>\lambda</td>
<td>λ</td>
<td>\mu</td>
<td>μ</td>
</tr>
<tr>
<td>\nu</td>
<td>ν</td>
<td>\xi</td>
<td>ξ</td>
</tr>
<tr>
<td>\pi</td>
<td>π</td>
<td>\rho</td>
<td>ρ</td>
</tr>
<tr>
<td>\sigma</td>
<td>σ</td>
<td>\tau</td>
<td>τ</td>
</tr>
<tr>
<td>\upsilon</td>
<td>υ</td>
<td>\phi</td>
<td>φ</td>
</tr>
<tr>
<td>\chi</td>
<td>χ</td>
<td>\psi</td>
<td>ψ</td>
</tr>
<tr>
<td>\omega</td>
<td>ω</td>
<td>\Gamma</td>
<td>Γ</td>
</tr>
<tr>
<td>\Delta</td>
<td>Δ</td>
<td>\Theta</td>
<td>Θ</td>
</tr>
<tr>
<td>\Lambda</td>
<td>Λ</td>
<td>\Xi</td>
<td>Ξ</td>
</tr>
<tr>
<td>\Pi</td>
<td>Π</td>
<td>\Sigma</td>
<td>Σ</td>
</tr>
<tr>
<td>\Upsilon</td>
<td>Υ</td>
<td>\Phi</td>
<td>Φ</td>
</tr>
<tr>
<td>\Psi</td>
<td>Ψ</td>
<td>\Omega</td>
<td>Ω</td>
</tr>
<tr>
<td>\bool</td>
<td>I B</td>
<td>\complex</td>
<td>C</td>
</tr>
<tr>
<td>\nat</td>
<td>I N</td>
<td>\rat</td>
<td>Q</td>
</tr>
<tr>
<td>\real</td>
<td>I R</td>
<td>\int</td>
<td>Z</td>
</tr>
<tr>
<td>\leftarrow</td>
<td>←</td>
<td>\longleftarrow</td>
<td>←−</td>
</tr>
<tr>
<td>\rightarrow</td>
<td>→</td>
<td>\longrightarrow</td>
<td>−→</td>
</tr>
<tr>
<td>\Leftarrow</td>
<td>⇐</td>
<td>\Longleftarrow</td>
<td>⇐−</td>
</tr>
<tr>
<td>\Rightarrow</td>
<td>⇒</td>
<td>\Longrightarrow</td>
<td>⇒−</td>
</tr>
<tr>
<td>\leftrightarrow</td>
<td>↔</td>
<td>\longleftrightarrow</td>
<td>←→</td>
</tr>
<tr>
<td>\Leftrightarrow</td>
<td>⇔</td>
<td>\Longleftrightarrow</td>
<td>⇔−</td>
</tr>
<tr>
<td>\mapsto</td>
<td>↦</td>
<td>\longmapsto</td>
<td>−→</td>
</tr>
<tr>
<td>\hookleftarrow</td>
<td>↩</td>
<td>\hookrightarrow</td>
<td>→</td>
</tr>
<tr>
<td>\leftharpoondown</td>
<td>↩</td>
<td>\rightharpoondown</td>
<td>→</td>
</tr>
<tr>
<td>\leftharpoonup</td>
<td>↩</td>
<td>\rightharpoonup</td>
<td>→</td>
</tr>
<tr>
<td>\rightleftharpoons</td>
<td>⇐⇒</td>
<td>\leadsto</td>
<td>;</td>
</tr>
<tr>
<td>\downharpoonleft</td>
<td>↓</td>
<td>\downharpoonright</td>
<td>↓</td>
</tr>
<tr>
<td>\upharpoonleft</td>
<td>↑</td>
<td>\upharpoonright</td>
<td>↑</td>
</tr>
</tbody>
</table>
APPENDIX B. PREDEFINED ISABELLE SYMBOLS

\texttt{\<restriction\>}  \[ \texttt{\<Colon\>}  \[ \texttt{\<up\>}  \[ \texttt{\<down\>}  \[ \texttt{\<updown\>}  \[ \texttt{\<langle\>}  \[ \texttt{\<ceil\>}  \[ \texttt{\<lfloor\>}  \[ \texttt{\<guillemotleft\>}  \[ \texttt{\<bottom\>}  \[ \texttt{\<and\>}  \[ \texttt{\<or\>}  \[ \texttt{\<forall\>}  \[ \texttt{\<box\>}  \[ \texttt{\<turnstile\>}  \[ \texttt{\<leq\>}  \[ \texttt{\<lless\>}  \[ \texttt{\<in\>}  \[ \texttt{\<subset\>}  \[ \texttt{\<inter\>}  \[ \texttt{\<union\>}  \[ \texttt{\<setminus\>}  \[ \texttt{\<uplus\>}  \[ \texttt{\<noteq\>}  \[ \texttt{\<approx\>}  \[ \texttt{\<cong\>}}


\begin{verbatim}
\textless equiv \textgreater \equiv
\textless Join \textgreater \Join
\textless prec \textgreater \prec
\textless preceq \textgreater \preceq
\textless parallel \textgreater \parallel
\textless plusminus \textgreater \pm
\textless times \textgreater \times
\textless cdot \textgreater \cdot
\textless bullet \textgreater \bullet
\textless dagger \textgreater \dagger
\textless 1hd \textgreater \langle
\textless unlhd \textgreater \rangle
\textless triangleleft \textgreater \triangleleft
\textless triangle \textgreater \triangle
\textless triangleq \textgreater \triangleq
\textless oplus \textgreater \oplus
\textless otimes \textgreater \otimes
\textless odot \textgreater \odot
\textless dots \textgreater \ldots
\textless Coprod \textgreater \coprod
\textless integral \textgreater \int
\textless clubsuit \textgreater \clubsuit
\textless heartsuit \textgreater \heartsuit
\textless aleph \textgreater \aleph
\textless nabla \textgreater \nabla
\textless Re \textgreater \Re
\textless flat \textgreater \flat
\textless sharp \textgreater \sharp
\textless copyright \textgreater \copyright
\textless hyphen \textgreater \hyphen
\textless onesuperior \textgreater \textless onequarter \textgreater \textless onehalf \textgreater
\textless twosuperior \textgreater \textless threequarters \textgreater
\textless ordfeminine \textgreater \textless ordmasculine \textgreater
\textless section \textgreater \textless paragraph \textgreater
\textless exclamdown \textgreater \textless questiondown \textgreater
\textless euro \textgreater \textless pounds \textgreater
\textless yen \textgreater \textless cent \textgreater
\end{verbatim}

\[
\begin{array}{ll}
\textlt{equiv} & \equiv \\
\textlt{Join} & \Join \\
\textlt{prec} & \preceq \\
\textlt{preceq} & \preceq \\
\textlt{parallel} & \parallel \\
\textlt{plusminus} & \pm \\
\textlt{times} & \times \\
\textlt{cdot} & \cdot \\
\textlt{bullet} & \bullet \\
\textlt{dagger} & \dagger \\
\textlt{1hd} & \langle \\
\textlt{unlhd} & \rangle \\
\textlt{triangleleft} & \triangleleft \\
\textlt{triangle} & \triangle \\
\textlt{oplus} & \oplus \\
\textlt{otimes} & \otimes \\
\textlt{odot} & \odot \\
\textlt{ominus} & \ominus \\
\textlt{dots} & \ldots \\
\textlt{Coprod} & \coprod \\
\textlt{integral} & \int \\
\textlt{clubsuit} & \clubsuit \\
\textlt{heartsuit} & \heartsuit \\
\textlt{aleph} & \aleph \\
\textlt{nabla} & \nabla \\
\textlt{Re} & \Re \\
\textlt{flat} & \flat \\
\textlt{sharp} & \sharp \\
\textlt{copyright} & \copyright \\
\textlt{hyphen} & \hyphen \\
\textlt{onesuperior} & \textless onequarter \\
\textlt{twosuperior} & \textless onehalf \\
\textlt{threesuperior} & \textless threequarters \\
\textlt{ordfeminine} & \textless ordmasculine \\
\textlt{section} & \textless paragraph \\
\textlt{exclamdown} & \textless questiondown \\
\textlt{euro} & \euro \\
\textlt{yen} & \yen \\
\end{array}
\]
\textbf{APPENDIX B. PREDEFINED ISABELLE SYMBOLS}

<table>
<thead>
<tr>
<th>Symbol</th>
<th>LaTeX Code</th>
<th>Unicode Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbullet</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>\textdegree</td>
<td>°</td>
<td>°</td>
</tr>
<tr>
<td>\textamalg</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>\textdegree</td>
<td>°</td>
<td>°</td>
</tr>
<tr>
<td>\textlozenge</td>
<td>◊</td>
<td>◊</td>
</tr>
<tr>
<td>\textdegree</td>
<td>°</td>
<td>°</td>
</tr>
<tr>
<td>\textmho</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>\textlozenge</td>
<td>◊</td>
<td>◊</td>
</tr>
<tr>
<td>\textdegree</td>
<td>°</td>
<td>°</td>
</tr>
<tr>
<td>\textstruct</td>
<td>◊</td>
<td>◊</td>
</tr>
<tr>
<td>\textdieresis</td>
<td>¨</td>
<td>¨</td>
</tr>
<tr>
<td>\textcedilla</td>
<td>¸</td>
<td>¸</td>
</tr>
<tr>
<td>\texthungarumlaut</td>
<td>&quot;</td>
<td>&quot;</td>
</tr>
<tr>
<td>\textmodule</td>
<td>⟨</td>
<td>⟨</td>
</tr>
<tr>
<td>\textbind</td>
<td>⇝</td>
<td>⇝</td>
</tr>
<tr>
<td>\textsome</td>
<td>ϵ</td>
<td>ϵ</td>
</tr>
</tbody>
</table>
ML tactic expressions

Isar Proof methods closely resemble traditional tactics, when used in unstructured sequences of apply commands. Isabelle/Isar provides emulations for all major ML tactics of classic Isabelle — mostly for the sake of easy porting of existing developments, as actual Isar proof texts would demand much less diversity of proof methods.

Unlike tactic expressions in ML, Isar proof methods provide proper concrete syntax for additional arguments, options, modifiers etc. Thus a typical method text is usually more concise than the corresponding ML tactic. Furthermore, the Isar versions of classic Isabelle tactics often cover several variant forms by a single method with separate options to tune the behavior. For example, method simp replaces all of simp_tac / asm_simp_tac / full_simp_tac / asm_full_simp_tac, there is also concrete syntax for augmenting the Simplifier context (the current “simpset”) in a convenient way.

C.1 Resolution tactics

Classic Isabelle provides several variant forms of tactics for single-step rule applications (based on higher-order resolution). The space of resolution tactics has the following main dimensions.

1. The “mode” of resolution: intro, elim, destruct, or forward (e.g. resolve_tac, eresolve_tac, dresolve_tac, forward_tac).
2. Optional explicit instantiation (e.g. resolve_tac vs. res_inst_tac).
3. Abbreviations for singleton arguments (e.g. resolve_tac vs. rtac).

Basically, the set of Isar tactic emulations rule_tac, erule_tac, drule_tac, frule_tac (see §9.2.3) would be sufficient to cover the four modes, either with or without instantiation, and either with single or multiple arguments. Although it is more convenient in most cases to use the plain rule method, or
any of its “improper” variants erule, drule, frule. Note that explicit goal addressing is only supported by the actual rule_tac version.

With this in mind, plain resolution tactics correspond to Isar methods as follows.

rtac a 1
resolve_tac [a1, ...] 1
res_inst_tac ctxt [(x1, t1), ...] a 1
rtac a i
resolve_tac [a1, ...] i
res_inst_tac ctxt [(x1, t1), ...] a i

Note that explicit goal addressing may be usually avoided by changing the order of subgoals with defer or prefer (see §6.3.4).

C.2 Simplifier tactics

The main Simplifier tactics simp_tac and variants (cf. [31]) are all covered by the simp and simp_all methods (see §9.3). Note that there is no individual goal addressing available, simplification acts either on the first goal (simp) or all goals (simp_all).

asm_full_simp_tac @{simpset} 1
ALLGOALS (asm_full_simp_tac @{simpset})
simp_tac @{simpset} 1
asm_simp_tac @{simpset} 1
full_simp_tac @{simpset} 1
asm_lr_simp_tac @{simpset} 1

C.3 Classical Reasoner tactics

The Classical Reasoner provides a rather large number of variations of automated tactics, such as blast_tac, fast_tac, clarify_tac etc. The corresponding Isar methods usually share the same base name, such as blast, fast, clarify etc. (see §9.4).
C.4 Miscellaneous tactics

There are a few additional tactics defined in various theories of Isabelle/HOL, some of these also in Isabelle/FOL or Isabelle/ZF. The most common ones of these may be ported to Isar as follows.

\begin{verbatim}
stac a 1 subst a
hyp_subst_tac 1 hypsubst
strip_tac 1 \approx intro strip
split_all_tac 1 simp \{no_asm_simp\} only: split_tupled_all
\approx simp only: split_tupled_all
\leq clarify
\end{verbatim}

C.5 Tacticals

Classic Isabelle provides a huge amount of tacticals for combination and modification of existing tactics. This has been greatly reduced in Isar, providing the bare minimum of combinators only: ";" (sequential composition), "|" (alternative choices), "?" (try), "+" (repeat at least once). These are usually sufficient in practice; if all fails, arbitrary ML tactic code may be invoked via the tactic method (see §9.2.3).

Common ML tacticals may be expressed directly in Isar as follows:

\begin{verbatim}
tac1 THEN tac2 meth1, meth2
tac1 ORELSE tac2 meth1 \mid meth2
TRY tac meth?
REPEAT1 tac meth+
REPEAT tac (meth+)?
EVERY [tac1, \ldots] meth1, \ldots
FIRST [tac1, \ldots] meth1 \mid \ldots
\end{verbatim}

CHANGED (see [43]) is usually not required in Isar, since most basic proof methods already fail unless there is an actual change in the goal state. Nevertheless, "?" (try) may be used to accept unchanged results as well.

ALLGOALS, SOMEGOAL etc. (see [43]) are not available in Isar, since there is no direct goal addressing. Nevertheless, some basic methods address all goals internally, notably simp_all (see §9.3). Also note that ALLGOALS can be often replaced by "+" (repeat at least once), although this usually has a different operational behavior: subgoals are solved in a different order.
Iterated resolution, such as `REPEAT (FIRSTGOAL (resolve_tac ...))`, is usually better expressed using the `intro` and `elim` methods of Isar (see §9.4).
Bibliography


275


http://isabelle.in.tum.de/doc/classes.pdf.


http://isabelle.in.tum.de/doc/logics-HOL.pdf.


Index

- (method), 120
  . (command), 118
  .. (command), 118
?thesis (variable), 116
  _ (fact), 112
  { (command), 107
  } (command), 107
abbrev (antiquote), 64
abbreviation (command), 81, 200
abs_def (attribute), 170
also (command), 127
altstring (syntax), 51, 59, 109
antiquote (syntax), 65
any (inner syntax), 153, 154
apply (command), 112, 113, 124
apply_end (command), 124
aprop (inner syntax), 153, 155
args (syntax), 58
arith (HOL attribute), 230
arith (HOL method), 230
arith_split (HOL attribute), 230
arities (command), 100
arity (syntax), 54
assms (fact), 113
assume (command), 108
assumes (element), 86
assumption (inference), 30
assumption (method), 120
atom (syntax), 58
atomize (attribute), 195
atomize (method), 195
attribute_setup (command), 97
attributes (syntax), 58
auto (method), 189
ax_specification (HOL command), 250
axiomatization (command), 81, 221
axioms (command), 102
axndecl (syntax), 59
back (command), 124
best (method), 189
bestsimp (method), 189
binder (keyword), 148
blast (method), 189
break (antiquote option), 69
bundle (command), 80
by (command), 118
calculation (fact), 127
case (command), 129
case_conclusion (attribute), 129
case_names (attribute), 129
case_tac (HOL method), 238
case_tac (ZF method), 257
cases (attribute), 137
cases (method), 116, 131, 132
cd (command), 166
chapter (command), 62
clamod (syntax), 190
clarify (method), 193
clarify_step_tac (ML), 194
clarsimp (method), 193
classimpmod (syntax), 191
class (antiquote), 64
class (command), 92
class_deps (command), 92
class_name (inner syntax), 154
INDEX

classdecl (syntax), 54
classes (command), 99
classrel (command), 99
cleaning (HOL method), 226
codatatype (ZF command), 255
code (HOL attribute), 241
code_abort (HOL command), 241
code_class (HOL command), 241
code_const (HOL command), 241
code_datatype (HOL command), 241
code_deps (HOL command), 241
code_include (HOL command), 241
code_instance (HOL command), 241
code_modulename (HOL command), 241
code_monad (HOL command), 241
code_post (HOL attribute), 241
code_pred (HOL command), 241
code_reflect (HOL command), 241
code_reserved (HOL command), 241
code_thms (HOL command), 241
code_type (HOL command), 241
code_unfold (HOL attribute), 241
coeq (HOL attribute), 229
coeq_enabled (HOL attribute), 229
coeq_map (HOL attribute), 229
coherent (HOL method), 232
coinduct (attribute), 137
coinduct (method), 132
coinductive (HOL command), 199
coinductive (ZF command), 255
coinductive_set (HOL command), 199
comment (syntax), 54
COMP (attribute), 170
cong (attribute), 178
const (antiquotation), 64
constrains (element), 86
consts (command), 101
consts (HOLCF command), 252
consumes (attribute), 129
context (command), 78
context_elem (syntax), 87
contradiction (method), 188
corollary (command), 113
cut_tac (method), 174
datatype (HOL command), 204, 213
datatype (ZF command), 255
declaration (command), 84
declare (command), 84
deepe (method), 189
def (command), 108
default_sort (command), 99
defer (command), 124
defines (element), 86
definition (command), 81
defn (attribute), 81
defs (command), 101
descending (HOL method), 226
descending_setup (HOL method), 226
dest (attribute), 187
dest (Pure attribute), 120
discharge (inference), 33
display (antiquotation option), 69
display_drafts (command), 75
domain (HOLCF command), 252
done (command), 124
drule (method), 168
drule_tac (method), 174
elem (attribute), 187
elem (method), 168
elem (Pure attribute), 120
elem_format (Pure attribute), 170
elem_resolution (inference), 30
end (global command), 76
end (local command), 78, 94
enriched_type (HOL command), 222
erule (method), 168
erule_tac (method), 174
eta_contract (antiquotation option), 69
eta_contract (attribute), 142
expansion (inference), 33
export_code (HOL command), 241
fact (method), 59, 120
fail (method), 168
fast (method), 189
fastforce (method), 189
file (antiquotation), 64
finally (command), 127
find_consts (command), 163
find_theorems (command), 163
find_unused_assms (HOL command), 234
finish (inference), 29
fix (command), 108
fixes (element), 86
fixes (syntax), 82
float (syntax), 51
float_const (inner syntax), 151
float_token (inner syntax), 151
fold (method), 168
folded (attribute), 170
for (keyword), 103
force (method), 189
from (command), 111
frule (method), 168
frule_tac (method), 174
full_prf (antiquotation), 64
full_prf (command), 139
fun (HOL command), 203
function (HOL command), 203
goal_spec (syntax), 118
goals (antiquotation), 64
goals_limit (antiquotation option), 69
goals_limit (attribute), 142
guess (command), 126
have (command), 113
header (command), 62
hence (command), 113
hide_class (command), 104
hide_const (command), 104
hide_fact (command), 104
hide_type (command), 104
hypsubst (method), 171
id (inner syntax), 151
ident (syntax), 50, 151
idt (inner syntax), 154, 155
idts (inner syntax), 154, 155
iff (attribute), 187
in (keyword), 79
include (command), 80
includes (keyword), 80
includes (syntax), 78, 80, 114
including (command), 80
ind_cases (HOL method), 238
ind_cases (ZF method), 257
indent (antiquotation option), 69
index (inner syntax), 154, 155
induct (attribute), 137
induct (method), 113, 131, 132
induct_simp (attribute), 136
induct_tac (HOL method), 238
induct_tac (ZF method), 257
induction (method), 132
induction_schema (HOL method), 208
inductive (HOL command), 199
inductive (ZF command), 255
inductive_cases (HOL command), 238
inductive_cases (ZF command), 257
inductive_set (HOL command), 199
infix (keyword), 148
infixl (keyword), 148
INDEX

infixr (keyword), 148
init (inference), 29
injection (HOL method), 226
insert (method), 168
inst (syntax), 55
inst_step_tac (ML), 194
instance (command), 92, 100
instantiation (command), 92, 101
insts (syntax), 55
int (syntax), 53
interpret (command), 89
interpretation (command), 89
intro (attribute), 187
intro (method), 168
intro (Pure attribute), 120
intro_classes (method), 92
intro_locales (method), 86
invariant_commute (HOL attribute), 224
iprover (HOL method), 230
is (keyword), 110
judgment (command), 195

keywords (keyword), 77

lemma (antiquotation), 64
lemma (command), 113
lemmas (command), 102
let (command), 110
lexicographic_order (HOL method), 208
lift_definition (HOL command), 224, 225
lifting (HOL method), 226
lifting_setup (HOL method), 226
local_setup (command), 97
locale (command), 86
locale (syntax), 87
locale_expr (syntax), 85
logic (inner syntax), 153, 155
longid (inner syntax), 151
longident (syntax), 50, 151

margin (antiquotation option), 69
meson (HOL method), 231
method (syntax), 117
method_setup (command), 125
metis (HOL method), 231
mixfix (syntax), 145
ML (antiquotation), 64
ML (command), 97
ML_command (command), 97
ML_op (antiquotation), 64
ML_prf (command), 97
ML_struct (antiquotation), 64
ML_type (antiquotation), 64
ML_val (command), 97
mode (antiquotation option), 69
modes (syntax), 140
mono (HOL attribute), 199
monos (ZF syntax), 255
moreover (command), 127

name (syntax), 52
nameref (syntax), 53
names_long (antiquotation option), 69
names_long (attribute), 142
names_short (antiquotation option), 69
names_short (attribute), 142
names_unique (antiquotation option), 69
names_unique (attribute), 142
nat (syntax), 51, 51, 151
next (command), 107
nitpick (HOL command), 234
nitpick_params (HOL command), 234
no_notation (command), 149
no_syntax (command), 158
no_translations (command), 158
INDEX

no_type_notation (command), 149
no_vars (attribute), 170
nonterminal (command), 158
notation (command), 149
note (command), 111
notepad (command), 106
notes (element), 86
nothing (fact), 112
num_const (inner syntax), 151
num_token (inner syntax), 151
obtain (command), 126
obtains (element), 113, 116
OF (attribute), 120
of (attribute), 120
oops (command), 108
oracle (command), 103
output (keyword), 160
overloading (command), 96
params (attribute), 129
parname (syntax), 52
parse_ast_translation (command), 161
parse_translation (command), 161
partial_function (HOL command), 210
partial_function_mono (HOL attribute), 210
partiality_descending (HOL method), 226
partiality_descending_setup (HOL method), 226
pat_completeness (HOL method), 208
pr (command), 139
prefer (command), 124
prems (fact), 110
presume (command), 108
Pretty.margin_default (ML), 145
prf (antiquotation), 64
primrec (HOL command), 203
primrec (ZF command), 257
print_abbrevs (command), 81
print_ast_translation (command), 161
print_attributes (command), 163
print_binds (command), 163
print_bundles (command), 80
print_cases (command), 129
print_claset (command), 187
print_classes (command), 92
print_codeproc (HOL command), 241
print_codesetup (HOL command), 241
print_commands (command), 163
print_configs (command), 167
print_dependencies (command), 89
print_depth (ML), 145
print_drafts (command), 75
print_facts (command), 163
print_induct_rules (command), 137
print_interps (command), 89
print_locale (command), 86
print_locales (command), 86
print_methods (command), 163
Print_Mode.with_modes (ML), 144
print_mode_value (ML), 144
print_quotconsts (HOL command), 226
print_quotients (HOL command), 224
print_quotientsQ3 (HOL command), 226
print_quotmaps (HOL command), 224
print_quotmapsQ3 (HOL command), 226
print_simpset (command), 178
print_statement (command), 113
INDEX

print_syntax (command), 157
print_tcsset (ZF command), 254
print_theorems (command), 163
print_theory (command), 163
print_trans_rules (command), 127
print_translation (command), 161
proof
    default, 120
    fake, 120
    terminal, 120
    trivial, 120
proof (command), 112, 113, 118, 118, 122
prop (antiquotation), 64
prop (command), 139
prop (inner syntax), 153, 155
prop (syntax), 55
prop_pat (syntax), 56
props (syntax), 57
pttrn (inner syntax), 154, 155
pttrns (inner syntax), 154, 155
pwd (command), 166
qed (command), 118, 118
quickcheck (HOL command), 234
quickcheck_generator (HOL command), 234
quickcheck_params (HOL command), 234
quot_lifted (HOL attribute), 226
quot_map (HOL attribute), 224
quot_preserve (HOL attribute), 226
quot_respect (HOL attribute), 226
quot_thm (HOL attribute), 226
quotes (antiquotation option), 69
quotient_definition (HOL command), 226
quotient_type (HOL command), 226
rail (antiquotation), 71
raw_tactic (method), 174
real (syntax), 53
recdef (HOL command), 211
recdef_cong (HOL attribute), 212
recdef_simp (HOL attribute), 212
recdef_tc (HOL command), 211
recdef_wf (HOL attribute), 212
record (HOL command), 216
refute (HOL command), 234
refute_params (HOL command), 234
regularize (HOL method), 226
relation (HOL method), 208
relator_eq (HOL attribute), 223
rename_tac (method), 174
rep_datatype (HOL command), 204, 213
resolution (inference), 30
rotate_tac (method), 174
rotated (attribute), 170
rule (attribute), 187
rule (method), 119, 188
rule (Pure attribute), 120
rule (Pure method), 112, 119, 120, 122, 271
rule_format (attribute), 195
rule_tac (method), 174
rulify (attribute), 195
safe (method), 193
safe_step_tac (ML), 194
schematic_corollary (command), 113
schematic_lemma (command), 113
schematic_theorem (command), 113
sect (command), 62
section (command), 62
setup (command), 97
setup_lifting (HOL command), 224
show (command), 109, 113, 118
show_abbrs (antiquotation option), 69
show_abbrs (attribute), 142
show_brackets (attribute), 142
<table>
<thead>
<tr>
<th>Index Item</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>show consts (attribute)</td>
<td>142</td>
</tr>
<tr>
<td>show_hyps (attribute)</td>
<td>142</td>
</tr>
<tr>
<td>show_main_goal (attribute)</td>
<td>142</td>
</tr>
<tr>
<td>show_question_marks  (attribute)</td>
<td>142</td>
</tr>
<tr>
<td>show_sorts (antiquotation option)</td>
<td>69</td>
</tr>
<tr>
<td>show_sorts (attribute)</td>
<td>142</td>
</tr>
<tr>
<td>show_structs (antiquotation option)</td>
<td>69</td>
</tr>
<tr>
<td>show_tags (attribute)</td>
<td>142</td>
</tr>
<tr>
<td>show_types (antiquotation option)</td>
<td>69</td>
</tr>
<tr>
<td>show_types (attribute)</td>
<td>142</td>
</tr>
<tr>
<td>shows (element)</td>
<td>113</td>
</tr>
<tr>
<td>simp (attribute)</td>
<td>178</td>
</tr>
<tr>
<td>simp (method)</td>
<td>176</td>
</tr>
<tr>
<td>simp_all (method)</td>
<td>176</td>
</tr>
<tr>
<td>simplified (attribute)</td>
<td>181</td>
</tr>
<tr>
<td>simpmod (syntax)</td>
<td>177</td>
</tr>
<tr>
<td>simp_proc_setup (command)</td>
<td>180</td>
</tr>
<tr>
<td>size_change (HOL method)</td>
<td>208</td>
</tr>
<tr>
<td>sledgehammer (HOL command)</td>
<td>232</td>
</tr>
<tr>
<td>sledgehammer_params (HOL com-</td>
<td></td>
</tr>
<tr>
<td>mand)</td>
<td></td>
</tr>
<tr>
<td>slow (method)</td>
<td>189</td>
</tr>
<tr>
<td>slow_step_tac (ML)</td>
<td>194</td>
</tr>
<tr>
<td>slowsimp (method)</td>
<td>189</td>
</tr>
<tr>
<td>solve_direct (HOL command)</td>
<td>232</td>
</tr>
<tr>
<td>sorry (command)</td>
<td>108, 118</td>
</tr>
<tr>
<td>sort (inner syntax)</td>
<td>154, 155</td>
</tr>
<tr>
<td>sort (syntax)</td>
<td>54</td>
</tr>
<tr>
<td>source (antiquotation option)</td>
<td>70</td>
</tr>
<tr>
<td>specification (HOL command)</td>
<td>250</td>
</tr>
<tr>
<td>split (attribute)</td>
<td>178</td>
</tr>
<tr>
<td>split (method)</td>
<td>171</td>
</tr>
<tr>
<td>split_format (HOL attribute)</td>
<td>219</td>
</tr>
<tr>
<td>standard (attribute)</td>
<td>170</td>
</tr>
<tr>
<td>step_tac (ML)</td>
<td>194</td>
</tr>
<tr>
<td>str_token (inner syntax)</td>
<td>151</td>
</tr>
<tr>
<td>string (syntax)</td>
<td>51, 51</td>
</tr>
<tr>
<td>struct_mixfix (syntax)</td>
<td>146</td>
</tr>
<tr>
<td>subclass (command)</td>
<td>92, 100</td>
</tr>
<tr>
<td>subgoal_tac (method)</td>
<td>174</td>
</tr>
<tr>
<td>subgoals (antiquotation)</td>
<td>64</td>
</tr>
<tr>
<td>sublocale (command)</td>
<td>89</td>
</tr>
<tr>
<td>sect (command)</td>
<td>62</td>
</tr>
<tr>
<td>subsect (command)</td>
<td>62</td>
</tr>
<tr>
<td>subsubsection (command)</td>
<td>62</td>
</tr>
<tr>
<td>succeed (method)</td>
<td>168</td>
</tr>
<tr>
<td>swapped (attribute)</td>
<td>187</td>
</tr>
<tr>
<td>symident (syntax)</td>
<td>50</td>
</tr>
<tr>
<td>syntax (command)</td>
<td>158</td>
</tr>
<tr>
<td>syntax_ambiguity_limit (attribute)</td>
<td>157</td>
</tr>
<tr>
<td>syntax_ambiguity_warning (at-</td>
<td></td>
</tr>
<tr>
<td>tribute)</td>
<td></td>
</tr>
<tr>
<td>syntax_declaration (command)</td>
<td>84</td>
</tr>
<tr>
<td>tactic (method)</td>
<td>174</td>
</tr>
<tr>
<td>tagged (attribute)</td>
<td>170</td>
</tr>
<tr>
<td>tags (syntax)</td>
<td>70</td>
</tr>
<tr>
<td>target (syntax)</td>
<td>79</td>
</tr>
<tr>
<td>TC (ZF attribute)</td>
<td>254</td>
</tr>
<tr>
<td>term (antiquotation)</td>
<td>64</td>
</tr>
<tr>
<td>term (command)</td>
<td>139</td>
</tr>
<tr>
<td>term (syntax)</td>
<td>55</td>
</tr>
<tr>
<td>term abbreviations,</td>
<td>111</td>
</tr>
<tr>
<td>term_pat (syntax)</td>
<td>56</td>
</tr>
<tr>
<td>term_type (antiquotation)</td>
<td>64</td>
</tr>
<tr>
<td>termination (HOL command)</td>
<td>203</td>
</tr>
<tr>
<td>text (antiquotation)</td>
<td>64</td>
</tr>
<tr>
<td>text (command)</td>
<td>62</td>
</tr>
<tr>
<td>text (syntax)</td>
<td>53</td>
</tr>
<tr>
<td>text_raw (command)</td>
<td>62</td>
</tr>
<tr>
<td>THEN (attribute)</td>
<td>170</td>
</tr>
<tr>
<td>then (command)</td>
<td>111, 113</td>
</tr>
<tr>
<td>theorem (command)</td>
<td>113</td>
</tr>
<tr>
<td>theorems (command)</td>
<td>102</td>
</tr>
<tr>
<td>theory (antiquotation)</td>
<td>64</td>
</tr>
<tr>
<td>theory (command)</td>
<td>76</td>
</tr>
</tbody>
</table>
thesis (variable), 111
thin_tac (method), 174
this (fact), 106, 111
this (method), 120
this (variable), 111
thm (antiquotation), 64
thm (command), 139
thm_deps (command), 163
thmdecl (syntax), 59
thmdef (syntax), 59
thmref (syntax), 59
thmrefs (syntax), 60
thus (command), 113
tid (inner syntax), 151
transfer (HOL method), 223
transfer’ (HOL method), 223
transfer_prover (HOL method), 223
transfer_rule (HOL attribute), 223
translations (command), 158
try (HOL command), 232
try0 (HOL command), 232
tvar (inner syntax), 151
txt (command), 62
txt_raw (command), 62
typ (antiquotation), 64
typ (command), 139
type (antiquotation), 64
type (inner syntax), 154, 155
type (syntax), 55
type_name (inner syntax), 154
type_notation (command), 149
type_synonym (command), 100, 219
typecheck (ZF method), 254
typed_print_translation (command), 161
typedecl (command), 100, 221
typedef (command), 213, 221
typedef (HOL command), 219
typeelims (ZF syntax), 256
typefree (syntax), 51, 151
typeintrs (ZF syntax), 256
typeof (antiquotation), 64
typespec (syntax), 56
typespec_sorts (syntax), 56
typevar (syntax), 51, 51, 151
ultimately (command), 127
unfold (method), 168
unfold_locales (method), 86
unfolded (attribute), 170
unfolding (command), 111
untagged (attribute), 170
unused_thms (command), 163
use (command), 78, 97
use_thy (command), 166
uses (keyword), 78, 98
using (command), 111
value (HOL command), 67, 234
values (HOL command), 234
var (inner syntax), 151
var (syntax), 51, 51, 151
vars (syntax), 57
verbatim (syntax), 51, 51
weak-discharge (inference), 33
where (attribute), 120
with (command), 111
write (command), 149
xnum_token (inner syntax), 151