

The Isabelle/Isar Reference Manual

Makarius Wenzel

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

November 15, 2026

Preface

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.

In its pioneering times, the Isabelle/Isar version of the *Proof General* user interface [2, 3] has contributed to the success of for interactive theory and proof development in this advanced theorem proving environment, even though it was somewhat biased towards old-style proof scripts. The more recent Isabelle/jEdit Prover IDE [61] emphasizes the document-oriented approach of Isabelle/Isar again more explicitly.

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 [58, 59]. *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 Isabelle/HOL are described in part III. Although the main language elements are already provided by the Isabelle/Pure framework, examples given in the generic parts will usually refer to Isabelle/HOL.

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**

1 Synopsis **2**

1.1 Notepad . . . 2
1.1.1 Types and terms . . . 2
1.1.2 Facts . . . 2
1.1.3 Block structure . . . 5
1.2 Calculational reasoning . . . 6
1.2.1 Special names in Isar proofs . . . 6
1.2.2 Transitive chains . . . 7
1.2.3 Degenerate calculations . . . 8
1.3 Induction . . . 9
1.3.1 Induction as Natural Deduction . . . 9
1.3.2 Induction with local parameters and premises . . . 11
1.3.3 Implicit induction context . . . 12
1.3.4 Advanced induction with term definitions . . . 12
1.4 Natural Deduction . . . 13
1.4.1 Rule statements . . . 13
1.4.2 Isar context elements . . . 14
1.4.3 Pure rule composition . . . 15
1.4.4 Structured backward reasoning . . . 16
1.4.5 Structured rule application . . . 17
1.4.6 Example: predicate logic . . . 18
1.5 Generalized elimination and cases . . . 21
1.5.1 General elimination rules . . . 21
1.5.2 Rules with cases . . . 22
1.5.3 Elimination statements and case-splitting . . . 24
1.5.4 Obtaining local contexts . . . 24

2 The Isabelle/Isar Framework **26**
2.1 The Pure framework 29
2.1.1 Primitive inferences 30
2.1.2 Reasoning with rules 30
2.2 The Isar proof language 32
2.2.1 Context elements 34
2.2.2 Structured statements 36
2.2.3 Structured proof refinement 37
2.2.4 Calculational reasoning 38
2.3 Example: First-Order Logic 39
2.3.1 Equational reasoning 40
2.3.2 Basic group theory 41
2.3.3 Propositional logic 42
2.3.4 Classical logic 44
2.3.5 Quantifiers 45
2.3.6 Canonical reasoning patterns 46

II General Language Elements **51**

3 Outer syntax — the theory language **52**
3.1 Commands 52
3.2 Lexical matters 53
3.3 Common syntax entities 55
3.3.1 Names 55
3.3.2 Numbers 56
3.3.3 Embedded content 56
3.3.4 Document text 57
3.3.5 Document comments 57
3.3.6 Type classes, sorts and arities 58
3.3.7 Types and terms 59
3.3.8 Term patterns and declarations 61
3.3.9 Attributes and theorems 63
3.3.10 Structured specifications 66
3.4 Diagnostic commands 67

4 Document preparation **71**
4.1 Markup commands . . . 71
4.2 Document antiquotations . . . 73
4.2.1 Styled antiquotations . . . 81
4.2.2 General options . . . 82
4.3 Markdown-like text structure . . . 83
4.4 Document markers and command tags . . . 84
4.5 Railroad diagrams . . . 87

5 Specifications **92**
5.1 Defining theories . . . 92
5.2 Local theory targets . . . 95
5.3 Bundled declarations . . . 97
5.4 Term definitions . . . 100
5.5 Axiomatizations . . . 102
5.6 Generic declarations . . . 103
5.7 Locales . . . 104
5.7.1 Locale expressions . . . 104
5.7.2 Locale declarations . . . 106
5.7.3 Locale interpretation . . . 110
5.8 Classes . . . 115
5.8.1 The class target . . . 118
5.8.2 Co-regularity of type classes and arities . . . 118
5.9 Overloaded constant definitions . . . 119
5.10 Overloaded constant abbreviations: adhoc overloading . . . 121
5.11 Incorporating ML code . . . 122
5.12 Generated files and exported files . . . 126
5.13 Primitive specification elements . . . 130
5.13.1 Sorts . . . 130
5.13.2 Types . . . 131
5.14 Naming existing theorems . . . 132
5.15 Oracles . . . 132
5.16 Name spaces . . . 133

6 Proofs **135**

6.1 Proof structure . . . 135
6.1.1 Formal notepad . . . 135
6.1.2 Blocks . . . 136
6.1.3 Omitting proofs . . . 137
6.2 Statements . . . 137
6.2.1 Context elements . . . 137
6.2.2 Term abbreviations . . . 139
6.2.3 Facts and forward chaining . . . 140
6.2.4 Goals . . . 142
6.3 Calculational reasoning . . . 146
6.4 Refinement steps . . . 148
6.4.1 Proof method expressions . . . 148
6.4.2 Initial and terminal proof steps . . . 150
6.4.3 Fundamental methods and attributes . . . 152
6.4.4 Defining proof methods . . . 156
6.5 Proof by cases and induction . . . 157
6.5.1 Rule contexts . . . 157
6.5.2 Proof methods . . . 160
6.5.3 Declaring rules . . . 165
6.6 Generalized elimination and case splitting . . . 166

7 Proof scripts 170
7.1 Commands for step-wise refinement . . . 170
7.2 Explicit subgoal structure . . . 172
7.3 Tactics: improper proof methods . . . 174

8 Inner syntax — the term language 178
8.1 Printing logical entities . . . 178
8.1.1 Diagnostic commands . . . 178
8.1.2 Details of printed content . . . 181
8.1.3 Alternative print modes . . . 183
8.2 Mixfix annotations . . . 184
8.2.1 The general mixfix form . . . 185
8.2.2 Infixes . . . 188
8.2.3 Binders . . . 189

8.3 Explicit notation . . . 190
8.4 The Pure syntax . . . 191
8.4.1 Lexical matters . . . 191
8.4.2 Priority grammars . . . 192
8.4.3 The Pure grammar . . . 193
8.4.4 Inspecting the syntax . . . 197
8.4.5 Ambiguity of parsed expressions . . . 198
8.5 Syntax transformations . . . 198
8.5.1 Abstract syntax trees . . . 199
8.5.2 Raw syntax and translations . . . 202
8.5.3 Syntax translation functions . . . 207
8.5.4 Built-in syntax transformations . . . 210

9 Generic tools and packages 213
9.1 Configuration options . . . 213
9.2 Basic proof tools . . . 214
9.2.1 Miscellaneous methods and attributes . . . 214
9.2.2 Low-level equational reasoning . . . 217
9.3 The Simplifier . . . 219
9.3.1 Simplification methods . . . 219
9.3.2 Declaring rules . . . 224
9.3.3 Ordered rewriting with permutative rules . . . 227
9.3.4 Simplifier tracing and debugging . . . 229
9.3.5 Simplification procedures . . . 231
9.3.6 Configurable Simplifier strategies . . . 234
9.3.7 Forward simplification . . . 238
9.4 The Classical Reasoner . . . 239
9.4.1 Basic concepts . . . 239
9.4.2 Rule declarations . . . 243
9.4.3 Structured methods . . . 245
9.4.4 Fully automated methods . . . 246
9.4.5 Partially automated methods . . . 250
9.4.6 Single-step tactics . . . 251
9.4.7 Modifying the search step . . . 252
9.5 Object-logic setup . . . 253

9.6 Tracing higher-order unification 255

III Isabelle/HOL 257

10 Higher-Order Logic 258

11 Derived specification elements 260
11.1 Inductive and coinductive definitions 260
11.1.1 Derived rules 262
11.1.2 Monotonicity theorems 262
11.2 Recursive functions 264
11.2.1 Proof methods related to recursive definitions 269
11.2.2 Functions with explicit partiality 271
11.2.3 Old-style recursive function definitions (TFL) 272
11.3 Definition by specification 273
11.4 Old-style datatypes 274
11.5 Records 275
11.5.1 Basic concepts 276
11.5.2 Record specifications 277
11.5.3 Record operations 278
11.5.4 Derived rules and proof tools 279
11.6 Semantic subtype definitions 280
11.7 Functorial structure of types 283
11.8 Quotient types with lifting and transfer 284
11.8.1 Quotient type definition 284
11.8.2 Lifting package 286
11.8.3 Transfer package 292
11.8.4 Old-style definitions for quotient types 294

12 Proof tools 297
12.1 Proving propositions 297
12.2 Checking and refuting propositions 299
12.3 Coercive subtyping 304
12.4 Arithmetic proof support 306
12.5 Intuitionistic proof search 306

12.6 Model Elimination and Resolution . . . 307
12.7 Algebraic reasoning via Gröbner bases . . . 307
12.8 Coherent Logic . . . 309
12.9 Unstructured case analysis and induction . . . 309
12.10Adhoc tuples . . . 311

13 Executable code **312**

IV Appendix **327**

A Isabelle/Isar quick reference **328**
A.1 Proof commands . . . 328
A.1.1 Main grammar . . . 328
A.1.2 Primitives . . . 329
A.1.3 Abbreviations and synonyms . . . 329
A.1.4 Derived elements . . . 329
A.1.5 Diagnostic commands . . . 330
A.2 Proof methods . . . 330
A.3 Attributes . . . 331
A.4 Rule declarations and methods . . . 331
A.5 Proof scripts . . . 332
A.5.1 Commands . . . 332
A.5.2 Methods . . . 332

B Predefined Isabelle symbols **333**

Bibliography **339**

Index **345**

List of Figures

2.1 Natural Deduction via inferences according to Gentzen, rules in Isabelle/Pure, and proofs in Isabelle/Isar 27
2.2 Main grammar of the Isar proof language 49
2.3 Isar/VM modes 50

8.1 Parsing and printing with translations 199

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

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):

define $x :: 'a$ **where** $x = t$

Abbreviations (polymorphic):

let *?f* $= \lambda x.\ x$
term *?f* *?f*

Notation:

write x ($\langle *** \rangle$)
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* ⟨*proof*⟩

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 ⋀*x*. *B x*
note ‹*B a*›
note ‹*B b*›
end

Manipulating facts

notepad
begin

Instantiation:

assume a: $\bigwedge x.\ B\ x$
note a
note a [*of* b]
note a [**where** $x = b$]

Backchaining:

assume 1: A
assume 2: $A \Longrightarrow C$
note 2 [*OF* 1]
note 1 [*THEN* 2]

Symmetric results:

assume $x = y$
note *this* [*symmetric*]

assume $x \neq y$
note *this* [*symmetric*]

Adhoc-simplification (take care!):

assume P ([] @ xs)
note *this* [*simplified*]
end

Projections

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

notepad
begin
assume *stuff*: $A\ B\ C\ D$
note *stuff*(1)
note *stuff*(2−3)
note *stuff*(2−)
end

Naming conventions

- Lower-case identifiers are usually preferred.
- Facts can be named after the main term within the proposition.
- Facts should *not* be named after the command that introduced them (**assume**, **have**). This is misleading and hard to maintain.

- Natural numbers can be used as "meaningless" names (more appropriate than $a1$, $a2$ 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* ⟨*proof*⟩
    have *b*: *B* ⟨*proof*⟩
    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* ⟨*proof*⟩
  next
    have *b*: *B*
    proof −
      show *B* ⟨*proof*⟩
    next
      have *c*: *C* ⟨*proof*⟩
    next
      have *d*: *D* ⟨*proof*⟩
    qed
  }

Alternative version with explicit parentheses everywhere:

```
{
  {
    have a: A ⟨proof⟩
  }
  {
    have b: B
    proof −
      {
        show B ⟨proof⟩
      }
      {
        have c: C ⟨proof⟩
      }
      {
        have d: D ⟨proof⟩
      }
    qed
  }
}

end
```

1.2 Calculational reasoning

For example, see `~~/src/HOL/Isar_Examples/Group.thy`.

1.2.1 Special names in Isar proofs

- term *?thesis* — the main conclusion of the innermost pending claim
- term ... — the argument of the last explicitly stated result (for infix application this is the right-hand side)
- fact *this* — the last result produced in the text

```
notepad
begin
  have x = y
  proof −
    term ?thesis
    show ?thesis ⟨proof⟩
    term ?thesis — static!
```

qed
term ...
thm *this*
end

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*

notepad
begin

Plain bottom-up calculation:

have $a = b$ ⟨*proof*⟩
also
have $b = c$ ⟨*proof*⟩
also
have $c = d$ ⟨*proof*⟩
finally
have $a = d$ **.**

Variant using the ... abbreviation:

have $a = b$ ⟨*proof*⟩
also
have ... $= c$ ⟨*proof*⟩
also
have ... $= d$ ⟨*proof*⟩
finally
have $a = d$ **.**

Top-down version with explicit claim at the head:

have $a = d$
proof −
  have $a = b$ ⟨*proof*⟩

```
    also
    have ... = c ⟨proof⟩
    also
    have ... = d ⟨proof⟩
    finally
    show ?thesis .
  qed
next
```

Mixed inequalities (require suitable base type):

```
  fix a b c d :: nat

  have a < b ⟨proof⟩
  also
  have b ≤ c ⟨proof⟩
  also
  have c = d ⟨proof⟩
  finally
  have a < d .
end
```

Notes

- The notion of *trans* rule is very general due to the flexibility of Isabelle/Pure rule composition.
- User applications may declare their own rules, with some care about the operational details of higher-order unification.

1.2.3 Degenerate calculations

The Idea is to append *this* to *calculation*, without rule composition. This is occasionally useful to avoid naming intermediate facts.

```
notepad
begin
```

A vacuous proof:

```
  have A ⟨proof⟩
  moreover
  have B ⟨proof⟩
  moreover
```

have C ⟨*proof*⟩
ultimately
have A **and** B **and** C **.**
next

Slightly more content (trivial bigstep reasoning):

have A ⟨*proof*⟩
moreover
have B ⟨*proof*⟩
moreover
have C ⟨*proof*⟩
ultimately
have $A \wedge B \wedge C$ **by** *blast*
end

Note that For multi-branch case splitting, it is better to use **consider**.

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:

thm *nat.induct*
print_statement *nat.induct*

notepad
begin
fix n :: *nat*
have $P\ n$
proof (*rule nat.induct*) — fragile rule application!
show $P\ 0$ ⟨*proof*⟩
next
fix n :: *nat*
assume $P\ n$
show $P\ (Suc\ n)$ ⟨*proof*⟩
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.

notepad
begin
  fix n :: *nat*
  have $P\ n$
  proof (*induct n*)
    case 0
    show *?case* ⟨*proof*⟩
  next
    case (*Suc n*)
    from *Suc.hyps* **show** *?case* ⟨*proof*⟩
  qed
end

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 $(\sum i{=}0..n.\ i) = n * (n + 1)\ div\ 2$
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 $\ldots = (n * (n + 1) + 2 * (n + 1))\ div\ 2$ **by** *simp*
  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 $P\ 0$ ⟨*proof*⟩
next
case (*Suc* n)
from ‹$P\ n$› **show** $P\ (Suc\ n)$ ⟨*proof*⟩
qed

have $A\ n \Longrightarrow P\ n$
proof (*induct* n)
case 0
from ‹$A\ 0$› **show** $P\ 0$ ⟨*proof*⟩
next
case (*Suc* n)
from ‹$A\ n \Longrightarrow P\ n$›
and ‹$A\ (Suc\ n)$› **show** $P\ (Suc\ n)$ ⟨*proof*⟩
qed

have $\bigwedge x.\ Q\ x\ n$
proof (*induct* n)
case 0
show $Q\ x\ 0$ ⟨*proof*⟩

```
  next
    case (Suc n)
    from ‹⋀x. Q x n› show Q x (Suc n) ⟨proof⟩
```

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 ⟨proof⟩
  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) ⟨proof⟩
  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.

```
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) ⟨proof⟩
  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) ⟨proof⟩
  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 $\bigwedge$ and $\Longrightarrow$ indicate the rule structure declaratively. For example:

```
thm conjI
thm impI
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 $(P \Longrightarrow \mathit{False}) \Longrightarrow \neg\ P$ **by** *blast*
lemma $\neg\ P \Longrightarrow P \Longrightarrow Q$ **by** *blast*

lemma $P \Longrightarrow Q \Longrightarrow P \wedge Q$ **by** *blast*
lemma $P \wedge Q \Longrightarrow (P \Longrightarrow Q \Longrightarrow R) \Longrightarrow R$ **by** *blast*

lemma $P \Longrightarrow P \vee Q$ **by** *blast*
lemma $Q \Longrightarrow P \vee Q$ **by** *blast*
lemma $P \vee Q \Longrightarrow (P \Longrightarrow R) \Longrightarrow (Q \Longrightarrow R) \Longrightarrow R$ **by** *blast*

lemma $(\bigwedge x.\ P\ x) \Longrightarrow (\forall x.\ P\ x)$ **by** *blast*
lemma $(\forall x.\ P\ x) \Longrightarrow P\ x$ **by** *blast*

lemma $P\ x \Longrightarrow (\exists x.\ P\ x)$ **by** *blast*
lemma $(\exists x.\ P\ x) \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow R) \Longrightarrow R$ **by** *blast*

lemma $x \in A \Longrightarrow x \in B \Longrightarrow x \in A \cap B$ **by** *blast*
lemma $x \in A \cap B \Longrightarrow (x \in A \Longrightarrow x \in B \Longrightarrow R) \Longrightarrow R$ **by** *blast*

lemma $x \in A \Longrightarrow x \in A \cup B$ **by** *blast*
lemma $x \in B \Longrightarrow x \in A \cup B$ **by** *blast*
lemma $x \in A \cup B \Longrightarrow (x \in A \Longrightarrow R) \Longrightarrow (x \in B \Longrightarrow R) \Longrightarrow R$ **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.

notepad
begin
 {
  fix x
  have $B\ x$ ⟨*proof*⟩
 }
 have $\bigwedge x.\ B\ x$ **by** *fact*

next

 {
  assume A
  have B ⟨*proof*⟩
 }
 have $A \Longrightarrow B$ **by** *fact*

next

 {
  define x **where** $x = t$
  have $B\ x$ ⟨*proof*⟩
 }
 have $B\ t$ **by** *fact*

next

 {
  obtain $x :: 'a$ **where** $B\ x$ ⟨*proof*⟩
  have C ⟨*proof*⟩
 }
 have C **by** *fact*

end

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 λ-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 / unmaintainable 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 ⋀x. A x ⟹ B x
  proof −
    fix x
    assume A x
    show B x ⟨proof⟩
  qed

  have ⋀x. A x ⟹ B x
  proof −
    {
      fix x
      assume A x
      show B x ⟨proof⟩
    } — implicit block structure made explicit
    note ‹⋀x. A x ⟹ B x›
      — side exit for the resulting rule
  qed
end
```

1.4.5 Structured rule application

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

notepad
begin
assume r_1: $A \Longrightarrow B \Longrightarrow C$ — simple rule (Horn clause)

have A ⟨*proof*⟩ — prefix of facts via outer sub-proof
then have C
proof (*rule* r_1)
show B ⟨*proof*⟩ — remaining rule premises via inner sub-proof
qed

have C
proof (*rule* r_1)
show A ⟨*proof*⟩
show B ⟨*proof*⟩
qed

have A **and** B ⟨*proof*⟩
then have C
proof (*rule* r_1)
qed

have A **and** B ⟨*proof*⟩
then have C **by** (*rule* r_1)

next

assume r_2: $A \Longrightarrow (\bigwedge x.\ B_1\ x \Longrightarrow B_2\ x) \Longrightarrow C$ — nested rule

have A ⟨*proof*⟩
then have C
proof (*rule* r_2)
fix x
assume $B_1\ x$
show $B_2\ x$ ⟨*proof*⟩
qed

The compound rule premise $\bigwedge x.\ B_1\ x \Longrightarrow B_2\ x$ 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 $A \longrightarrow B$ **and** A ⟨*proof*⟩
then have B **..**

have A ⟨*proof*⟩
then have $A \vee B$ **..**

have B ⟨*proof*⟩
then have $A \vee B$ **..**

have $A \vee B$ ⟨*proof*⟩
then have C
proof
assume A
then show C ⟨*proof*⟩
next
assume B
then show C ⟨*proof*⟩
qed

have A **and** B ⟨*proof*⟩
then have $A \wedge B$ **..**

have $A \wedge B$ ⟨*proof*⟩
then have A **..**

have $A \wedge B$ ⟨*proof*⟩
then have B **..**

have *False* ⟨*proof*⟩
then have A **..**

have *True* **..**

have $\neg\ A$
proof
assume A
then show *False* ⟨*proof*⟩

qed

have $\neg\ A$ **and** A ⟨*proof*⟩
then have B **..**

have $\forall x.\ P\ x$
proof
fix x
show $P\ x$ ⟨*proof*⟩
qed

have $\forall x.\ P\ x$ ⟨*proof*⟩
then have $P\ a$ **..**

have $\exists x.\ P\ x$
proof
show $P\ a$ ⟨*proof*⟩
qed

have $\exists x.\ P\ x$ ⟨*proof*⟩
then have C
proof
fix a
assume $P\ a$
show C ⟨*proof*⟩
qed

Less awkward version using **obtain**:

have $\exists x.\ P\ x$ ⟨*proof*⟩
then obtain a **where** $P\ a$ **..**
end

Further variations to illustrate Isar sub-proofs involving **show**:

notepad
begin
have $A \wedge B$
proof — two strictly isolated subproofs
show A ⟨*proof*⟩
next
show B ⟨*proof*⟩
qed

have $A \wedge B$

```
  proof — one simultaneous sub-proof
    show A and B ⟨proof⟩
  qed

  have A ∧ B
  proof — two subproofs in the same context
    show A ⟨proof⟩
    show B ⟨proof⟩
  qed

  have A ∧ B
  proof — swapped order
    show B ⟨proof⟩
    show A ⟨proof⟩
  qed

  have A ∧ B
  proof — sequential subproofs
    show A ⟨proof⟩
    show B using ‹A› ⟨proof⟩
  qed
end
```

Example: set-theoretic operators

There is nothing special about logical connectives ($\wedge$, $\vee$, $\forall$, $\exists$ 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 ∈ A and x ∈ B ⟨proof⟩
  then have x ∈ A ∩ B ..

  have x ∈ A ⟨proof⟩
  then have x ∈ A ∪ B ..

  have x ∈ B ⟨proof⟩
  then have x ∈ A ∪ B ..

  have x ∈ A ∪ B ⟨proof⟩
  then have C
  proof
    assume x ∈ A
```

```
    then show C ⟨proof⟩
  next
    assume x ∈ B
    then show C ⟨proof⟩
  qed

next
  have x ∈ ⋂A
  proof
    fix a
    assume a ∈ A
    show x ∈ a ⟨proof⟩
  qed

  have x ∈ ⋂A ⟨proof⟩
  then have x ∈ a
  proof
    show a ∈ A ⟨proof⟩
  qed

  have a ∈ A and x ∈ a ⟨proof⟩
  then have x ∈ ⋃A ..

  have x ∈ ⋃A ⟨proof⟩
  then obtain a where a ∈ A and x ∈ a ..
end
```

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:

thm *exE* — local parameter
thm *conjE* — local premises
thm *disjE* — 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

notepad
begin
assume r:
$A_1 \Longrightarrow A_2 \Longrightarrow$ — assumptions
$(\bigwedge x\ y.\ B_1\ x\ y \Longrightarrow C_1\ x\ y \Longrightarrow R) \Longrightarrow$ — case 1
$(\bigwedge x\ y.\ B_2\ x\ y \Longrightarrow C_2\ x\ y \Longrightarrow R) \Longrightarrow$ — case 2
R — main conclusion

have A_1 **and** A_2 ⟨*proof*⟩
then have R
proof (*rule r*)
fix $x\ y$
assume $B_1\ x\ y$ **and** $C_1\ x\ y$
show *?thesis* ⟨*proof*⟩
next
fix $x\ y$
assume $B_2\ x\ y$ **and** $C_2\ x\ y$
show *?thesis* ⟨*proof*⟩
qed
end

Here *?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 A_1 **and** A_2 — assumptions
obtains
($case_1$) x y **where** B_1 x y **and** C_1 x y
| ($case_2$) x y **where** B_2 x y **and** C_2 x y
⟨*proof*⟩

Example

lemma *tertium_non_datur*:
obtains
(T) A
| (F) $\neg$ 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* ⟨*proof*⟩
next
case F
from ‹$x \neq y$› **show** *?thesis* ⟨*proof*⟩
qed
end

Example

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

datatype *foo* = *Foo* | *Bar foo*

notepad
begin
fix x :: *foo*
have C
proof (*cases* x)
case *Foo*
from ‹x = *Foo*› **show** *?thesis* ⟨*proof*⟩
next
case (*Bar a*)

```
    from ‹x = Bar a› show ?thesis ⟨proof⟩
  qed
end
```

1.5.3 Elimination statements and case-splitting

The **consider** states rules for generalized elimination and case splitting. This is like a toplevel statement **theorem obtains** used within a proof body; or like a multi-branch **obtain** without activation of the local context elements yet.

The proof method *cases* is able to use such rules with forward-chaining (e.g. via **then**). This leads to the subsequent pattern for case-splitting in a particular situation within a proof.

```
notepad
begin
  consider (a) A | (b) B | (c) C
    ⟨proof⟩ — typically by auto, by blast etc.
  then have something
  proof cases
    case a
    then show ?thesis ⟨proof⟩
  next
    case b
    then show ?thesis ⟨proof⟩
  next
    case c
    then show ?thesis ⟨proof⟩
  qed
end
```

1.5.4 Obtaining local contexts

A single "case" branch may be inlined into Isar proof text via **obtain**. This proves $(\bigwedge x.\ B\ x \Longrightarrow \mathit{thesis}) \Longrightarrow \mathit{thesis}$ on the spot, and augments the context afterwards.

```
notepad
begin
  fix B :: 'a ⇒ bool

  obtain x where B x ⟨proof⟩
  note ‹B x›
```

Conclusions from this context may not mention x again!

{
  obtain x **where** $B\ x$ ⟨*proof*⟩
  from ‹$B\ x$› **have** C ⟨*proof*⟩
}
note ‹C›
end

Chapter 2

The Isabelle/Isar Framework

Isabelle/Isar [58, 59, 37, 63, 62, 60] is a generic framework for developing formal mathematical documents with full proof checking. Definitions, statements and proofs are organized as theories. A collection of theories sources may be presented as a printed document; see also chapter 4.

The main concern 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.

Thus 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). Technically, Isar is 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 commands may later appear as an intelligible text to the human reader.

The Isar proof language has emerged from careful analysis of some inherent virtues of the logical framework Isabelle/Pure [45, 46], notably composition of higher-order natural deduction rules, which is a generalization of Gentzen's original calculus [18]. The approach of generic inference systems in Pure is continued by Isar towards actual proof texts. See also figure 2.1

Concrete applications require another intermediate layer: an object-logic. Isabelle/HOL [39] (simply-typed set-theory) is most commonly used; elementary examples are given in the directories `~~/src/Pure/Examples` and `~~/src/HOL/Examples`. Some examples demonstrate how to start a fresh object-logic from Isabelle/Pure, and use Isar proofs from the very start, despite the lack of advanced proof tools at such an early stage (e.g. see `~~/src/Pure/Examples/Higher_Order_Logic.thy`). Isabelle/FOL [42] and Isabelle/ZF [43] also work, but are much less developed.

Inferences:

$$\frac{A \longrightarrow B \quad A}{B} \qquad \frac{\begin{array}{c}[A]\\ \vdots \\ B\end{array}}{A \longrightarrow B}$$

Isabelle/Pure:

$$(A \longrightarrow B) \Longrightarrow A \Longrightarrow B \qquad (A \Longrightarrow B) \Longrightarrow A \longrightarrow B$$

Isabelle/Isar:

have $A \longrightarrow B$ $\langle proof \rangle$	**have** $A \longrightarrow B$
also have A $\langle proof \rangle$	**proof**
finally have B **.**	**assume** A
	then show B $\langle proof \rangle$
	qed

Figure 2.1: Natural Deduction via inferences according to Gentzen, rules in Isabelle/Pure, and proofs in Isabelle/Isar

In order to illustrate natural deduction in Isar, we shall subsequently 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 ($\wedge$, $\vee$, $\forall$, $\exists$, 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 fact and goal statements against rules that are declared in the library context.

assume $x \in A$ **and** $x \in B$
then have $x \in A \cap B$ **..**

$$\frac{x \in A \quad x \in B}{x \in A \cap B}$$

Note that **assume** augments the proof context, **then** indicates that the cur-

rent fact shall be used in the next step, and **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 **by** command:

assume $x \in A$ **and** $x \in B$
then have $x \in A \cap B$ **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 of $\bigcap \mathcal{A}$, 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:

have $x \in \bigcap \mathcal{A}$
proof
  fix A
  assume $A \in \mathcal{A}$
  show $x \in A$ $\langle$*proof*$\rangle$
qed

$$\frac{\begin{array}{c}[A][A \in \mathcal{A}] \\ \vdots \\ x \in A\end{array}}{x \in \bigcap \mathcal{A}}$$

This Isar reasoning pattern again refers to the primitive rule depicted above. The system determines it in the "**proof**" step, which could have been spelled 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. Such compound rules typically demands a genuine subproof in Isar, working backwards rather than forwards as seen before. In the proof body we encounter the **fix-assume-show** outline of nested subproofs 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 $\exists A.\ x \in A \wedge A \in \mathcal{A}$. The elimination rule for $x \in \bigcup \mathcal{A}$ does not mention $\exists$ and $\wedge$ 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:

assume $x \in \bigcup \mathcal{A}$
then have C
proof
  fix A
  assume $x \in A$ **and** $A \in \mathcal{A}$
  show C $\langle proof \rangle$
qed

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

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:

assume $x \in \bigcup \mathcal{A}$
then obtain A **where** $x \in A$ **and** $A \in \mathcal{A}$ **..**

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 [45, 46] is an intuitionistic fragment of higher-order logic [14]. In type-theoretic parlance, there are three levels of λ-calculus with corresponding arrows $\Rightarrow/\bigwedge/\Longrightarrow$:

$\alpha \Rightarrow \beta$	syntactic function space (terms depending on terms)
$\bigwedge x.\ B(x)$	universal quantification (proofs depending on terms)
$A \Longrightarrow B$	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 [6], 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 [52]. Here object-logic inferences are internalized as formulae over $\bigwedge$ and $\Longrightarrow$. Combining such rule statements may involve higher-order unification [44].

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 $\bigwedge x :: \alpha.\ B(x)$ and $A \Longrightarrow B$. Primitive reasoning operates on judgments of the form $\Gamma \vdash \varphi$, with standard introduction and elimination rules for $\bigwedge$ and $\Longrightarrow$ that refer to fixed parameters $x_1, \ldots, x_m$ and hypotheses $A_1, \ldots, A_n$ from the context Γ; the corresponding proof terms are left implicit. The subsequent inference rules define $\Gamma \vdash \varphi$ inductively, relative to a collection of axioms from the implicit background theory:

$$\frac{A \text{ is axiom}}{\vdash A} \qquad \frac{}{A \vdash A}$$

$$\frac{\Gamma \vdash B(x) \quad x \notin \Gamma}{\Gamma \vdash \bigwedge x.\ B(x)} \qquad \frac{\Gamma \vdash \bigwedge x.\ B(x)}{\Gamma \vdash B(a)}$$

$$\frac{\Gamma \vdash B}{\Gamma - A \vdash A \Longrightarrow B} \qquad \frac{\Gamma_1 \vdash A \Longrightarrow B \quad \Gamma_2 \vdash A}{\Gamma_1 \cup \Gamma_2 \vdash B}$$

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

An object-logic introduces another layer on top of Pure, e.g. with types i for individuals and o for propositions, term constants $\mathit{Trueprop} :: o \Rightarrow \mathit{prop}$ as (implicit) derivability judgment and connectives like $\wedge :: o \Rightarrow o \Rightarrow o$ or $\forall :: (i \Rightarrow o) \Rightarrow o$, and axioms for object-level rules such as conjI: $A \Longrightarrow B \Longrightarrow A \wedge B$ or allI: $(\bigwedge x.\ B\ x) \Longrightarrow \forall x.\ B\ x$. Derived object rules are represented as theorems of Pure. After the initial object-logic setup, further axiomatizations are usually avoided: definitional principles are used instead (e.g. **definition**, **inductive**, **fun**, **function**).

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 $\bigwedge$ to bind local parameters and $\Longrightarrow$ to express entailment. Multiple parameters and premises are represented by repeating these connectives in a right-associative manner.

Thanks to the Pure theorem $(A \Longrightarrow (\bigwedge x.\ B\ x)) \equiv (\bigwedge x.\ A \Longrightarrow B\ x)$ the connectives $\bigwedge$ and $\Longrightarrow$ commute. So 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* [33] which is of the form $\bigwedge x_1\ \ldots\ x_m.\ H_1 \Longrightarrow \ldots\ H_n \Longrightarrow A$ 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 $A_1 \Longrightarrow \ldots\ A_n \Longrightarrow A$ are a special case of this.

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

$$\mathit{IntI}\colon\ x \in A \Longrightarrow x \in B \Longrightarrow x \in A \cap B$$

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

$$\mathit{InterI}\colon\ (\bigwedge A.\ A \in \mathcal{A} \Longrightarrow x \in A) \Longrightarrow x \in \bigcap \mathcal{A}$$

Goals are also represented as rules: $A_1 \Longrightarrow \ldots\ A_n \Longrightarrow C$ states that the subgoals A_1, $\ldots$, A_n entail the result C; for $n = 0$ the goal is finished. To allow C being a rule statement itself, there is an internal protective marker $\# :: \mathit{prop} \Rightarrow \mathit{prop}$, which is defined as identity and hidden from the user. We initialize and finish goal states as follows:

$$\frac{}{C \Longrightarrow \#C}\ (\mathit{init}) \qquad \frac{\#C}{C}\ (\mathit{finish})$$

Goal states are refined in intermediate proof steps until a finished form is achieved. Here the two main reasoning principles are *resolution*, for backchaining a rule against a subgoal (replacing it by zero or more subgoals), and *assumption*, for solving a subgoal (finding a short-circuit with local assumptions). Below $\overline{x}$ stands for x_1, $\ldots$, x_n (for $n \geq 0$).

$$\frac{\begin{array}{rl} \mathit{rule}\colon & \overline{A}\ \overline{a} \Longrightarrow B\ \overline{a} \\ \mathit{goal}\colon & (\bigwedge \overline{x}.\ \overline{H}\ \overline{x} \Longrightarrow B'\ \overline{x}) \Longrightarrow C \\ \mathit{goal\ unifier}\colon & (\lambda \overline{x}.\ B\ (\overline{a}\ \overline{x}))\,\theta = B'\theta \end{array}}{(\bigwedge \overline{x}.\ \overline{H}\ \overline{x} \Longrightarrow \overline{A}\ (\overline{a}\ \overline{x}))\,\theta \Longrightarrow C\,\theta}\ (\mathit{resolution})$$

$$\frac{\begin{array}{rl} \textit{goal}: & (\bigwedge \overline{x}.\ \overline{H}\ \overline{x} \Longrightarrow A\ \overline{x}) \Longrightarrow C \\ \textit{assm unifier}: & A\,\theta = H_i\,\theta \ \text{ for some } H_i \end{array}}{C\,\theta}\ (\textit{assumption})$$

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

$$\begin{array}{rll} (A \wedge B \Longrightarrow B \wedge A) \Longrightarrow \#(A \wedge B \Longrightarrow B \wedge A) & (\textit{init}) \\ (A \wedge B \Longrightarrow B) \Longrightarrow (A \wedge B \Longrightarrow A) \Longrightarrow \#\ldots & (\textit{resolution}\ B \Longrightarrow A \Longrightarrow B \wedge A) \\ (A \wedge B \Longrightarrow A \wedge B) \Longrightarrow (A \wedge B \Longrightarrow A) \Longrightarrow \#\ldots & (\textit{resolution}\ A \wedge B \Longrightarrow B) \\ (A \wedge B \Longrightarrow A) \Longrightarrow \#\ldots & (\textit{assumption}) \\ (A \wedge B \Longrightarrow A \wedge B) \Longrightarrow \#\ldots & (\textit{resolution}\ A \wedge B \Longrightarrow A) \\ \#\ldots & (\textit{assumption}) \\ A \wedge B \Longrightarrow B \wedge A & (\textit{finish}) \end{array}$$

Compositions of *assumption* after *resolution* occurs quite often, typically in elimination steps. Traditional Isabelle tactics accommodate this by a combined *elim_resolution* principle. In contrast, Isar uses a combined refinement rule as follows:[1]

$$\frac{\begin{array}{rl} \textit{subgoal}: & (\bigwedge \overline{x}.\ \overline{H}\ \overline{x} \Longrightarrow B'\ \overline{x}) \Longrightarrow C \\ \textit{subproof}: & \overline{G}\ \overline{a} \Longrightarrow B\ \overline{a} \quad \text{for schematic } \overline{a} \\ \textit{concl unifier}: & (\lambda \overline{x}.\ B\ (\overline{a}\ \overline{x}))\,\theta = B'\theta \\ \textit{assm unifiers}: & (\lambda \overline{x}.\ G_j\ (\overline{a}\ \overline{x}))\,\theta = H_i\,\theta \quad \text{for each } G_j \text{ some } H_i \end{array}}{C\,\theta}\ (\textit{refinement})$$

Here the *subproof* rule stems from the main **fix-assume-show** outline of Isar (cf. §2.2.3): each assumption indicated in the text results in a marked premise G above. Consequently, **fix-assume-show** enables to fit the result of a subproof quite robustly into a pending subgoal, while maintaining a good measure of flexibility: the subproof only needs to fit modulo unification, and its assumptions may be a proper subset of the subgoal premises (see §2.2.3).

2.2 The Isar proof language

Structured proofs are presented as high-level expressions for composing entities of Pure (propositions, facts, and goals). The Isar proof language allows to organize reasoning within the underlying rule calculus of Pure, but Isar

[1] For simplicity and clarity, the presentation ignores *weak premises* as introduced via **presume** or **show** ... **when**.

is not another logical calculus. Isar merely imposes certain structure and policies on Pure inferences. The main grammar of the Isar proof language is given in figure 2.2.

The construction of the Isar proof language proceeds in a bottom-up fashion, as an exercise in purity and minimalism. The grammar in appendix A.1.1 describes the primitive parts of the core language (category *proof*), which is embedded into the main outer theory syntax via elements that require a proof (e.g. **theorem**, **lemma**, **function**, **termination**).

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

Facts may be referenced by name or proposition. For example, the result of "**have** *a*: *A* ⟨*proof*⟩" becomes accessible both via the name *a* and the literal proposition ‹*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**, there are some abbreviations:

from *a* ≡ **note** *a* **then**
with *a* ≡ **from** *a* **and** *this*

The *method* category is essentially a parameter of the Isar language and may be populated later. The command **method_setup** allows to define proof methods semantically in Isabelle/ML. The Eisbach language allows to define proof methods symbolically, as recursive expressions over existing methods [32]; see also `~~/src/HOL/Eisbach`.

Methods use the facts indicated by **then** or **using**, and then operate on the goal state. Some basic methods are predefined in Pure: "−" 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 "*standard*" (which subsumes *rule* with argu-

ments picked from the context), for **qed** it is "*succeed*". Further abbreviations for terminal proof steps are "**by** *method*$_1$ *method*$_2$" for "**proof** *method*$_1$ **qed** *method*$_2$", and "**..**" for "**by** *standard*, and "**.**" for "**by** *this*". The command "**unfolding** *facts*" operates directly on the goal by applying equalities.

Block structure can be indicated explicitly by "**{** ... **}**", although the body of a subproof "**proof** ... **qed**" already provides implicit nesting. In both situations, **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 connection to subgoals here!

The commands **fix** and **assume** build up a local context (see §2.2.1), while **show** refines a pending subgoal by the rule resulting from a nested subproof (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, Γ essentially acts like a proof context. Isar elaborates this idea towards a more advanced concept, 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 **assume** «*inference*» element provides a general interface to hypotheses: **assume** «*inference*» 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 surface syntax for «*inference*», i.e. it may only occur internally when derived commands are defined in ML.

The default inference for **assume** is *export* as given below. The derived element **define** x **where** $x \equiv a$ is defined as **fix** x **assume** «*expand*» $x \equiv a$, with the subsequent inference *expand*.

$$\frac{\Gamma \vdash B}{\Gamma - A \vdash A \Longrightarrow B}\,(\mathit{export}) \qquad \frac{\Gamma \vdash B\ x}{\Gamma - (x \equiv a) \vdash B\ a}\,(\mathit{expand})$$

The most interesting derived context element in Isar is **obtain** [59, §5.3], which supports generalized elimination steps in a purely forward manner. The **obtain** command takes a specification of parameters $\overline{x}$ 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):

⟨*facts*⟩ **obtain** $\overline{x}$ **where** $\overline{A}\ \overline{x}$ ⟨*proof*⟩ ≡
  have *case*: $\bigwedge$*thesis*. ($\bigwedge\overline{x}$. $\overline{A}\ \overline{x} \Longrightarrow$ *thesis*) $\Longrightarrow$ *thesis*
  proof −
    fix *thesis*
    assume [*intro*]: $\bigwedge\overline{x}$. $\overline{A}\ \overline{x} \Longrightarrow$ *thesis*
    show *thesis* **using** ⟨*facts*⟩ ⟨*proof*⟩
  qed
  fix $\overline{x}$ **assume** «*elimination case*» $\overline{A}\ \overline{x}$

$$\frac{\begin{array}{rl} \textit{case}: & \Gamma \vdash \bigwedge\textit{thesis}.\ (\bigwedge\overline{x}.\ \overline{A}\ \overline{x} \Longrightarrow \textit{thesis}) \Longrightarrow \textit{thesis} \\ \textit{result}: & \Gamma \cup \overline{A}\ \overline{y} \vdash B \end{array}}{\Gamma \vdash B}\ (\textit{elimination})$$

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** $A\ x$" can 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 subgoals that need to be proven separately.

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, the exported result is indicated via **note**.

{
  fix x
  have $B\ x$ ⟨*proof*⟩
}
note ‹$\bigwedge x.\ B\ x$›

{
  assume A
  have B ⟨*proof*⟩
}
note ‹$A \Longrightarrow B$›

{
  define x **where** $x \equiv a$
  have $B\ x$ ⟨*proof*⟩
}
note ‹$B\ a$›

{
  obtain x **where** $A\ x$ ⟨*proof*⟩
  have B ⟨*proof*⟩
}
note ‹B›

This explains the meaning of Isar context elements without, without goal states getting in the way.

2.2.2 Structured statements

The syntax of top-level theorem statements is defined as follows:

statement	≡	*name*: *props* **and** ...
	|	*context** *conclusion*
context	≡	**fixes** *vars* **and** ...
	|	**assumes** *name*: *props* **and** ...
conclusion	≡	**shows** *name*: *props* **and** ...
	|	**obtains** *vars* **and** ... **where** *name*: *props* **and** ...
		| ...

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: $\bigwedge x.\ A\ x \Longrightarrow B\ x$.

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.

obtains $\overline{x}$ **where** $\overline{A}\ \overline{x}$ | ... ≡
  fixes *thesis*
  assumes [*intro*]: $\bigwedge \overline{x}.\ \overline{A}\ \overline{x} \Longrightarrow$ *thesis* **and** ...
  shows *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 **shows** and **obtains**, respectively:

theorem
  fixes x **and** y
  assumes $A\ x$ **and** $B\ y$
  shows $C\ x\ y$
proof −
  from ‹$A\ x$› **and** ‹$B\ y$›
  show $C\ x\ y$ ⟨*proof*⟩
qed

theorem
  obtains x **and** y
  where $A\ x$ **and** $B\ y$
proof −
  have $A\ a$ **and** $B\ b$ ⟨*proof*⟩
  then show *thesis* **..**
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 "**then show** *thesis* **..**" involves the local rule case $\bigwedge x\ y.\ A\ x \Longrightarrow B\ y \Longrightarrow$ *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 2.3).

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 subgoals. 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:

have $A \longrightarrow B$	*begin*	*prove*	$(A \longrightarrow B) \Longrightarrow \#(A \longrightarrow B)$	(*init*)
proof		*state*	$(A \Longrightarrow B) \Longrightarrow \#(A \longrightarrow B)$	(*resolution impI*)
assume A		*state*		
show B	*begin*	*prove*		
$\langle$*proof*$\rangle$	*end*	*state*	$\#(A \longrightarrow B)$	(*refinement* $\#A \Longrightarrow B$)
qed	*end*	*state*	$A \longrightarrow B$	(*finish*)

Here the *refinement* inference from §2.1.2 mediates composition of Isar subproofs 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 subproof may be omitted altogether. For example:

have $\bigwedge x\ y.\ A\ x \Longrightarrow B\ y \Longrightarrow C\ x\ y$
proof −
  fix x **and** y
  assume $A\ x$ **and** $B\ y$
  show $C\ x\ y$ ⟨*proof*⟩
qed

have $\bigwedge x\ y.\ A\ x \Longrightarrow B\ y \Longrightarrow C\ x\ y$
proof −
  fix x **assume** $A\ x$
  fix y **assume** $B\ y$
  show $C\ x\ y$ ⟨*proof*⟩
qed

have $\bigwedge x\ y.\ A\ x \Longrightarrow B\ y \Longrightarrow C\ x\ y$
proof −
  fix y **assume** $B\ y$
  fix x **assume** $A\ x$
  show $C\ x\ y$ ⟨*proof*⟩
qed

have $\bigwedge x\ y.\ A\ x \Longrightarrow B\ y \Longrightarrow C\ x\ y$
proof −
  fix y **assume** $B\ y$
  fix x
  show $C\ x\ y$ ⟨*proof*⟩
qed

Such fine-tuning of Isar text is practically important to improve readability. Contexts elements are rearranged 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 $=$, $<$, $\leq$, $\subset$, $\subseteq$ 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 [59, §6] and [5].

The generic calculational mechanism is based on the observation that rules such as *trans*: $x = y \Longrightarrow y = z \Longrightarrow x = z$ 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 context:

$$\begin{array}{rcl} \textbf{also}_0 & \equiv & \textbf{note}\ \mathit{calculation} = \mathit{this} \\ \textbf{also}_{n+1} & \equiv & \textbf{note}\ \mathit{calculation} = \mathit{trans}\ [\mathit{OF}\ \mathit{calculation}\ \mathit{this}] \\ \textbf{finally} & \equiv & \textbf{also from}\ \mathit{calculation} \end{array}$$

The start of a calculation is determined implicitly in the text: here **also** sets *calculation* to the current result; any subsequent occurrence will update *calculation* by combination with the next result and a transitivity rule. The calculational sequence is concluded via **finally**, where the final result is exposed for use in a concluding claim.

Here is a canonical proof pattern, using **have** to establish the intermediate results:

have $a = b$ ⟨*proof*⟩
also have $\ldots = c$ ⟨*proof*⟩
also have $\ldots = d$ ⟨*proof*⟩
finally have $a = d$ **.**

The term "..." (literal ellipsis) is a special abbreviation provided by the Isabelle/Isar term syntax: 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 $x = y \Longrightarrow y = x$ are like transitivities with only one premise. Isar maintains a separate rule collection declared via the *sym* attribute, to be used in fact expressions "*a* [*symmetric*]", or single-step proofs "**assume** $x = y$ **then have** $y = x$ **..**".

2.3 Example: First-Order Logic

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.

typedecl *i*

typedecl *o*

judgment *Trueprop* :: $o \Rightarrow$ *prop* (‹_› 5)

Note that the object-logic judgment is implicit in the syntax: writing A produces *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 *equal* :: $i \Rightarrow i \Rightarrow o$ (**infix** ‹=› 50)
where *refl* [*intro*]: $x = x$
and *subst* [*elim*]: $x = y \Longrightarrow B\ x \Longrightarrow B\ y$

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

theorem *sym* [*sym*]:
assumes $x = y$
shows $y = x$
proof −
have $x = x$ **..**
with ‹$x = y$› **show** $y = x$ **..**
qed

theorem *forw_subst* [*trans*]:
assumes $y = x$ **and** $B\ x$
shows $B\ y$
proof −
from ‹$y = x$› **have** $x = y$ **..**
from *this* **and** ‹$B\ x$› **show** $B\ y$ **..**
qed

theorem *back_subst* [*trans*]:
assumes $B\ x$ **and** $x = y$
shows $B\ y$
proof −

```
  from ‹x = y› and ‹B x›
  show B y ..
qed

theorem trans [trans]:
  assumes x = y and y = z
  shows x = z
proof −
  from ‹y = z› and ‹x = y›
  show x = z ..
qed
```

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 group =
  fixes prod :: i ⇒ i ⇒ i  (infix ‹○› 70)
    and inv :: i ⇒ i  (‹(_⁻¹)› [1000] 999)
    and unit :: i  (‹1›)
  assumes assoc: (x ○ y) ○ z = x ○ (y ○ z)
    and left_unit:  1 ○ x = x
    and left_inv: x⁻¹ ○ x = 1
begin

theorem right_inv: x ○ x⁻¹ = 1
proof −
  have x ○ x⁻¹ = 1 ○ (x ○ x⁻¹) by (rule left_unit [symmetric])
  also have ... = (1 ○ x) ○ x⁻¹ by (rule assoc [symmetric])
  also have 1 = (x⁻¹)⁻¹ ○ x⁻¹ by (rule left_inv [symmetric])
  also have ... ○ x = (x⁻¹)⁻¹ ○ (x⁻¹ ○ x) by (rule assoc)
  also have x⁻¹ ○ x = 1 by (rule left_inv)
  also have ((x⁻¹)⁻¹ ○ ...) ○ x⁻¹ = (x⁻¹)⁻¹ ○ (1 ○ x⁻¹) by (rule assoc)
  also have 1 ○ x⁻¹ = x⁻¹ by (rule left_unit)
  also have (x⁻¹)⁻¹ ○ ... = 1 by (rule left_inv)
  finally show x ○ x⁻¹ = 1 .
qed

theorem right_unit: x ○ 1 = x
proof −
  have 1 = x⁻¹ ○ x by (rule left_inv [symmetric])
  also have x ○ ... = (x ○ x⁻¹) ○ 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** *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 [18].

axiomatization *imp* :: $o \Rightarrow o \Rightarrow o$ (**infixr** ‹$\longrightarrow$› 25)
where *impI* [*intro*]: $(A \Longrightarrow B) \Longrightarrow A \longrightarrow B$
and *impD* [*dest*]: $(A \longrightarrow B) \Longrightarrow A \Longrightarrow B$

axiomatization *disj* :: $o \Rightarrow o \Rightarrow o$ (**infixr** ‹$\vee$› 30)
where $disjI_1$ [*intro*]: $A \Longrightarrow A \vee B$
and $disjI_2$ [*intro*]: $B \Longrightarrow A \vee B$

and *disjE* [*elim*]: $A \vee B \Longrightarrow (A \Longrightarrow C) \Longrightarrow (B \Longrightarrow C) \Longrightarrow C$

axiomatization *conj* :: $o \Rightarrow o \Rightarrow o$ (**infixr** ‹$\wedge$› 35)
where *conjI* [*intro*]: $A \Longrightarrow B \Longrightarrow A \wedge B$
and $conjD_1$: $A \wedge B \Longrightarrow A$
and $conjD_2$: $A \wedge B \Longrightarrow B$

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

theorem *conjE* [*elim*]:
assumes $A \wedge B$
obtains A **and** B
proof
from ‹$A \wedge B$› **show** A **by** (*rule* $conjD_1$)
from ‹$A \wedge B$› **show** B **by** (*rule* $conjD_2$)
qed

Here is an example of swapping conjuncts with a single intermediate elimination step:

assume $A \wedge B$
then obtain B **and** A **..**
then have $B \wedge A$ **..**

Note that the analogous elimination rule for disjunction "**assumes** $A \vee B$ **obtains** $A \mid B$" coincides with the original axiomatization of *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 *false* :: o (‹$\bot$›)
where *falseE* [*elim*]: $\bot \Longrightarrow A$

definition *true* :: o (‹$\top$›)
where $\top \equiv \bot \longrightarrow \bot$

theorem *trueI* [*intro*]: $\top$
unfolding *true_def* **..**

Now negation represents an implication towards absurdity:

definition *not* :: $o \Rightarrow o$ (‹$\neg$ _› [40] 40)

```
  where ¬ A ≡ A ⟶ ⊥

theorem notI [intro]:
  assumes A ⟹ ⊥
  shows ¬ A
unfolding not_def
proof
  assume A
  then show ⊥ by (rule ‹A ⟹ ⊥›)
qed

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.

```
locale classical =
  assumes classical: (¬ C ⟹ C) ⟹ C
begin

theorem double_negation:
  assumes ¬ ¬ C
  shows C
proof (rule classical)
  assume ¬ C
  with ‹¬ ¬ C› show C ..
qed

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 $(\neg\ C \Longrightarrow C) \Longrightarrow C$ 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 [14], quantifiers are operators on predicates, which are syntactically represented as λ-terms of type $i \Rightarrow o$. Binder notation turns $All\ (\lambda x.\ B\ x)$ into $\forall x.\ B\ x$ etc.

```
axiomatization All :: (i ⇒ o) ⇒ o  (binder ‹∀› 10)
  where allI [intro]: (⋀x. B x) ⟹ ∀x. B x
    and allD [dest]: (∀x. B x) ⟹ B a

axiomatization Ex :: (i ⇒ o) ⇒ o  (binder ‹∃› 10)
  where exI [intro]: B a ⟹ (∃x. B x)
    and exE [elim]: (∃x. B x) ⟹ (⋀x. B x ⟹ C) ⟹ C
```

The statement of *exE* corresponds to "**assumes** $\exists x.\ B\ x$ **obtains** x **where** $B\ x$" in Isar. In the subsequent example we illustrate quantifier reasoning involving all four rules:

```
theorem
  assumes ∃x. ∀y. R x y
```

shows $\forall\, y.\ \exists\, x.\ R\ x\ y$
proof — $\forall$ introduction
obtain x **where** $\forall\, y.\ R\ x\ y$ **using** ‹$\exists\, x.\ \forall\, y.\ R\ x\ y$› **..** — $\exists$ elimination
fix y **have** $R\ x\ y$ **using** ‹$\forall\, y.\ R\ x\ y$› **..** — $\forall$ destruction
then show $\exists\, x.\ R\ x\ y$ **..** — $\exists$ introduction
qed

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.

impI: **assumes** $A \Longrightarrow B$ **shows** $A \longrightarrow B$
impD: **assumes** $A \longrightarrow B$ **and** A **shows** B

$disjI_1$: **assumes** A **shows** $A \vee B$
$disjI_2$: **assumes** B **shows** $A \vee B$
disjE: **assumes** $A \vee B$ **obtains** $A \mid B$

conjI: **assumes** A **and** B **shows** $A \wedge B$
conjE: **assumes** $A \wedge B$ **obtains** A **and** B

falseE: **assumes** $\bot$ **shows** A
trueI: **shows** $\top$

notI: **assumes** $A \Longrightarrow \bot$ **shows** $\neg\ A$
notE: **assumes** $\neg\ A$ **and** A **shows** B

allI: **assumes** $\bigwedge x.\ B\ x$ **shows** $\forall\, x.\ B\ x$
allE: **assumes** $\forall\, x.\ B\ x$ **shows** $B\ a$

exI: **assumes** $B\ a$ **shows** $\exists\, x.\ B\ x$
exE: **assumes** $\exists\, x.\ B\ x$ **obtains** a **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 *intro*, *elim*, *dest* — each according to its particular shape — we can immediately write Isar proof texts as follows:

have $A \longrightarrow B$
proof
  assume A
  show B ⟨*proof*⟩
qed

have $A \longrightarrow B$ **and** A ⟨*proof*⟩
then have B **..**

have A $\langle proof \rangle$
then have $A \vee B$ **..**

have B $\langle proof \rangle$
then have $A \vee B$ **..**

have $A \vee B$ $\langle proof \rangle$
then have C
proof
  assume A
  then show C $\langle proof \rangle$
next
  assume B
  then show C $\langle proof \rangle$
qed

have A **and** B $\langle proof \rangle$
then have $A \wedge B$ **..**

have $A \wedge B$ $\langle proof \rangle$
then obtain A **and** B **..**

have $\bot$ $\langle proof \rangle$
then have A **..**

have $\top$ **..**

have $\neg\ A$
proof
  assume A
  then show $\bot$ $\langle proof \rangle$
qed

have $\neg\ A$ **and** A $\langle proof \rangle$
then have B **..**

have $\forall x.\ B\ x$
proof
  fix x
  show $B\ x$ $\langle proof \rangle$
qed

have $\forall x.\ B\ x$ $\langle proof \rangle$
then have $B\ a$ **..**

have $\exists x.\ B\ x$
proof
  show $B\ a$ $\langle proof \rangle$
qed

have $\exists x.\ B\ x$ $\langle proof \rangle$
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

main = **notepad begin** *statement** **end**
| **theorem** *name*: *props* **if** *name*: *props* **for** *vars*
| **theorem** *name*:
fixes *vars*
assumes *name*: *props*
shows *name*: *props proof*
| **theorem** *name*:
fixes *vars*
assumes *name*: *props*
obtains (*name*) *clause* | ... *proof*

proof = *refinement** *proper_proof*

refinement = **apply** *method*
| **supply** *name* = *thms*
| **subgoal premises** *name* **for** *vars proof*
| **using** *thms*
| **unfolding** *thms*

proper_proof = **proof** *method*$^?$ *statement** **qed** *method*$^?$
| **by** *method method* | **..** | **.** | **sorry** | **done**

statement = **{** *statement** **}** | **next**
| **note** *name* = *thms*
| **let** *term* = *term*
| **write** *name* (*mixfix*)
| **fix** *vars*
| **assume** *name*: *props* **if** *props* **for** *vars*
| **presume** *name*: *props* **if** *props* **for** *vars*
| **define** *clause*
| **case** *name*: *case*
| **then**$^?$ *goal*
| **from** *thms goal*
| **with** *thms goal*
| **also** | **finally** *goal*
| **moreover** | **ultimately** *goal*

goal = **have** *name*: *props* **if** *name*: *props* **for** *vars proof*
| **show** *name*: *props* **if** *name*: *props* **for** *vars proof*
| **show** *name*: *props* **when** *name*: *props* **for** *vars proof*
| **consider** (*name*) *clause* | ... *proof*
| **obtain** (*name*) *clause proof*

clause = *vars* **where** *name*: *props* **if** *props* **for** *vars*

Figure 2.2: Main grammar of the Isar proof language

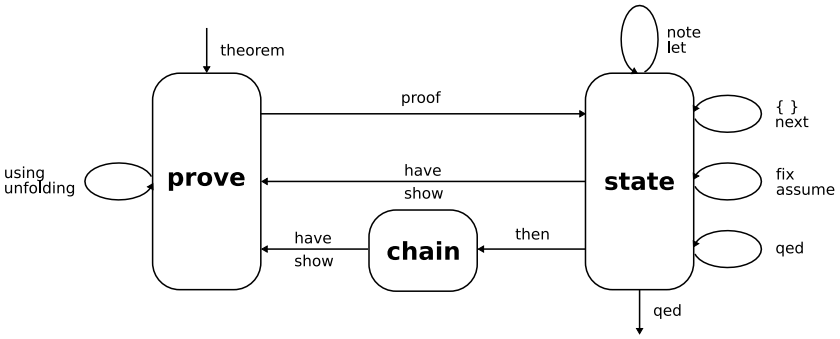


Figure 2.3: Isar/VM modes

Part II

General Language Elements

Chapter 3

Outer syntax — the theory language

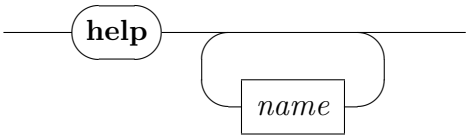
The rather generic framework of Isabelle/Isar syntax emerges from three main syntactic categories: *commands* of the top-level Isar engine (covering theory and proof elements), *methods* for general goal refinements (analogous to traditional "tactics"), and *attributes* for operations on facts (within 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 LaTeX macro setup, say via `\isabellestyle`, see also [54]). Experienced users of Isabelle/Isar may easily reconstruct the lost technical information, while mere readers need not care about quotes at all.

3.1 Commands

print_commands* : *any* $\to$
help* : *any* $\to$



print_commands prints all outer syntax keywords and commands.

help *pats* retrieves outer syntax commands according to the specified name patterns.

Examples

Some common diagnostic commands are retrieved like this (according to usual naming conventions):

help *print*
help *find*

3.2 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.

short_ident = *letter* (*subscript*? *quasiletter*)*
long_ident = *short_ident*(. *short_ident*)+
sym_ident = *sym*+ | \<*short_ident*>
nat = *digit*+
float = *nat* . *nat* | -*nat* . *nat*
term_var = ?*short_ident* | ?*short_ident* . *nat*
type_ident = ’*short_ident*
type_var = ?*type_ident* | ?*type_ident* . *nat*
string = " ... "
altstring = ‘ ... ‘
cartouche = \<open> ... \<close>
verbatim = {* ... *}

letter = *latin* | \<*latin*> | \<*latin latin*> | *greek* |
subscript = \<^sub>
quasiletter = *letter* | *digit* | _ | ’
latin = a | ... | z | A | ... | Z
digit = 0 | ... | 9
sym = ! | # | $ | % | & | * | + | - | / |
< | = | > | ? | @ | ^ | _ | | | ~
greek = \<alpha> | \<beta> | \<gamma> | \<delta> |
\<epsilon> | \<zeta> | \<eta> | \<theta> |
\<iota> | \<kappa> | \<mu> | \<nu> |
\<xi> | \<pi> | \<rho> | \<sigma> | \<tau> |
\<upsilon> | \<phi> | \<chi> | \<psi> |
\<omega> | \<Gamma> | \<Delta> | \<Theta> |
\<Lambda> | \<Xi> | \<Pi> | \<Sigma> |
\<Upsilon> | \<Phi> | \<Psi> | \<Omega>

A *term_var* or *type_var* describes an unknown, which is internally a pair of base name and index (ML type `indexname`). These components are either separated by a dot as in *?x*.1 or *?x*7.3 or run together as in *?x*1. The latter form is possible if the base name does not end with digits. If the index is 0, it may be dropped altogether: *?x* and *?x*0 and *?x*.0 all refer to the same unknown, with basename *x* and index 0.

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

The body of *verbatim* may consist of any text not containing "*}"; this allows

to include quotes without further escapes, but there is no way to escape "`*}`". Cartouches do not have this limitation.

A *cartouche* consists of arbitrary text, with properly balanced blocks of "`\<open> ... \<close>`". Note that the rendering of cartouche delimiters is usually like this: "‹ ... ›".

Source comments take the form `(* ... *)` and may be nested: the text is removed after lexical analysis of the input and thus not suitable for documentation. The Isar syntax also provides proper *document comments* that are considered as part of the text (see §3.3.5).

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

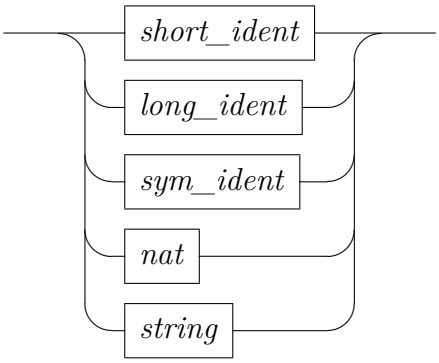
3.3 Common syntax entities

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.

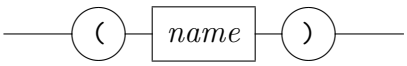
3.3.1 Names

Entity *name* usually refers to any name of types, constants, theorems etc. Quoted strings provide an escape for non-identifier names or those ruled out by outer syntax keywords (e.g. quoted `"let"`).

name



par_name

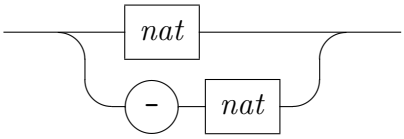


A *system_name* is like *name*, but it excludes white-space characters and needs to conform to file-name notation. Name components that are special on Windows (e.g. `CON`, `PRN`, `AUX`) are excluded on all platforms.

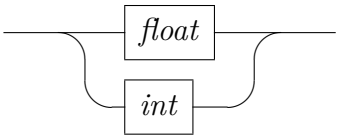
3.3.2 Numbers

The outer lexical syntax (§3.2) admits natural numbers and floating point numbers. These are combined as *int* and *real* as follows.

int



real

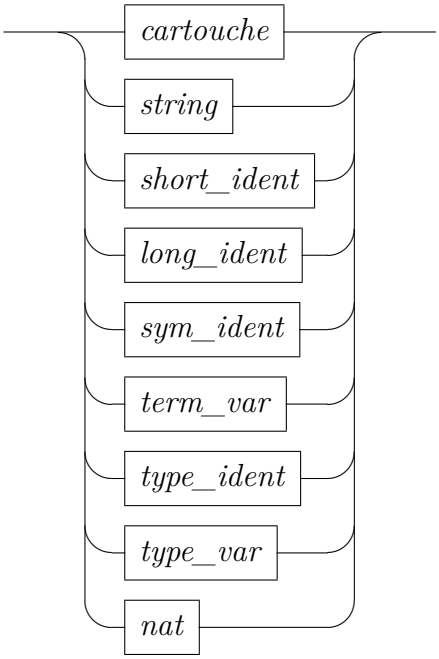


Note that there is an overlap with the category *name*, which also includes *nat*.

3.3.3 Embedded content

Entity *embedded* refers to content of other languages: cartouches allow arbitrary nesting of sub-languages that respect the recursive balancing of cartouche delimiters. Quoted strings are possible as well, but require escaped quotes when nested. As a shortcut, tokens that appear as plain identifiers in the outer language may be used as inner language content without delimiters.

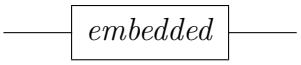
embedded



3.3.4 Document text

A chunk of document *text* is usually given as *cartouche* ‹. . . ›. For convenience, any of the smaller text unit that conforms to *name* is admitted as well.

text



Typical uses are document markup commands, like **chapter**, **section** etc. (§4.1).

3.3.5 Document comments

Formal comments are an integral part of the document, but are logically void and removed from the resulting theory or term content. The output of document preparation (chapter 4) supports various styles, according to the following kinds of comments.

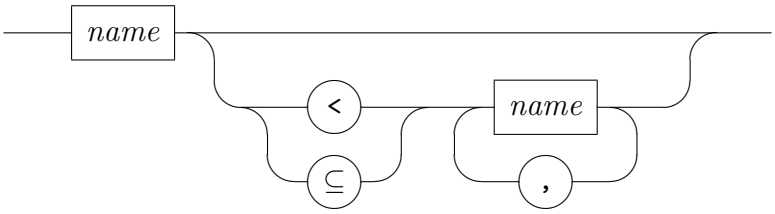
- Marginal comment of the form `\<comment>` ‹*text*› or — ‹*text*›, usually with a single space between the comment symbol and the argument cartouche. The given argument is typeset as regular text, with formal antiquotations (§4.2).
- Canceled text of the form `\<^cancel>`‹*text*› (no white space between the control symbol and the argument cartouche). The argument is typeset as formal Isabelle source and overlaid with a "strike-through" pattern, e.g. ~~*foo*~~.
- Raw LaTeX source of the form `\<^latex>`‹*argument*› (no white space between the control symbol and the argument cartouche). This allows to augment the generated TeX source arbitrarily, without any sanity checks!

These formal comments work uniformly in outer syntax, inner syntax (term language), Isabelle/ML, and some other embedded languages of Isabelle.

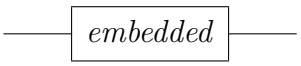
3.3.6 Type classes, sorts and arities

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.

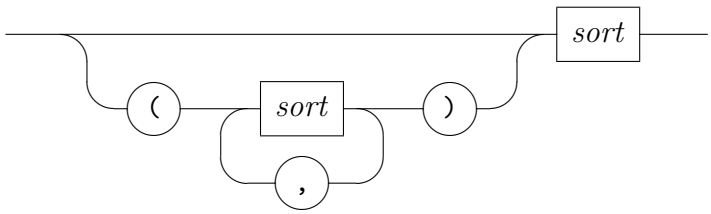
classdecl



sort



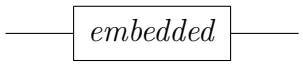
arity



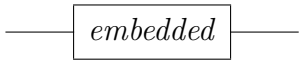
3.3.7 Types and terms

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 `+`).

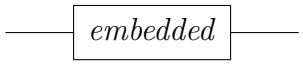
type



term

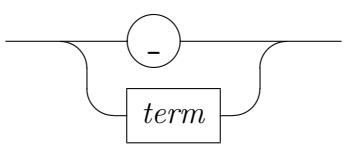


prop

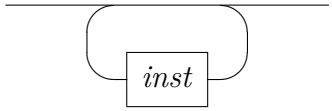


Positional instantiations are specified as a sequence of terms, or the placeholder "_" (underscore), which means to skip a position.

inst

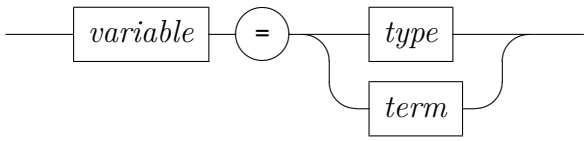


insts

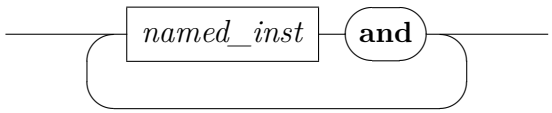


Named instantiations are specified as pairs of assignments $v = t$, which refer to schematic variables in some theorem that is instantiated. Both type and terms instantiations are admitted, and distinguished by the usual syntax of variable names.

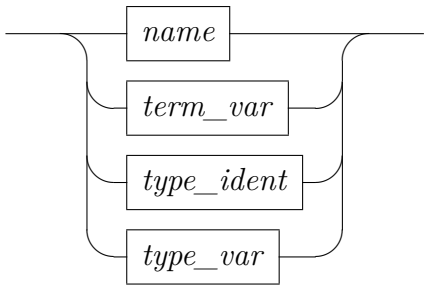
named_inst



named_insts

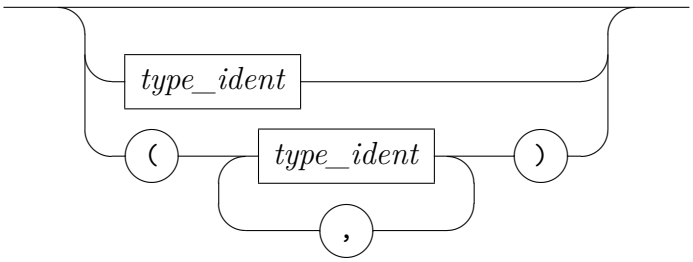


variable

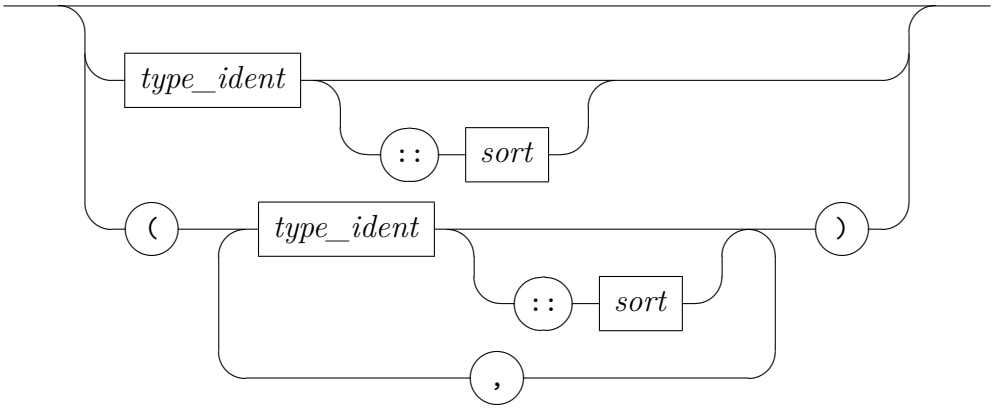


Type declarations and definitions usually refer to *typespec* on the left-hand side. This models basic type constructor application at the outer syntax level. Note that only plain postfix notation is available here, but no infixes.

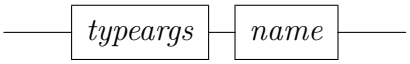
typeargs



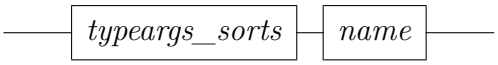
typeargs_sorts



typespec



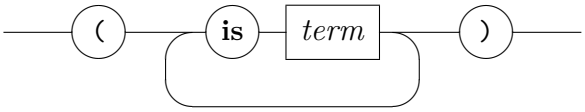
typespec_sorts



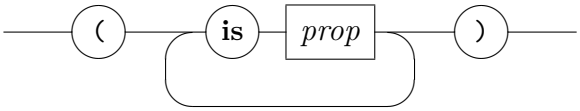
3.3.8 Term patterns and declarations

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 ... p_n)". This works both for *term* and *prop*.

term_pat

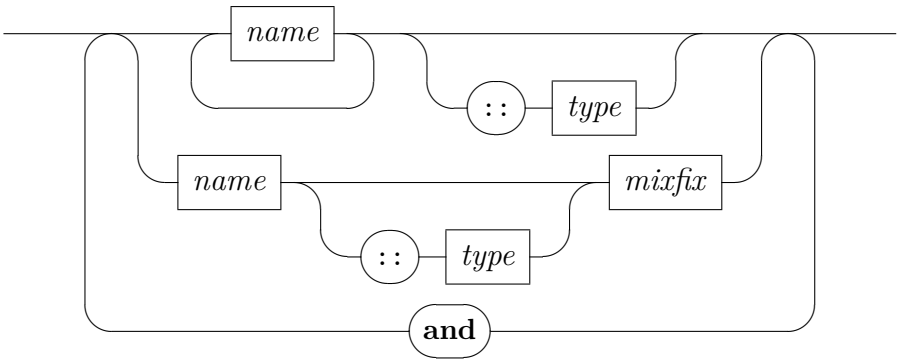


prop_pat

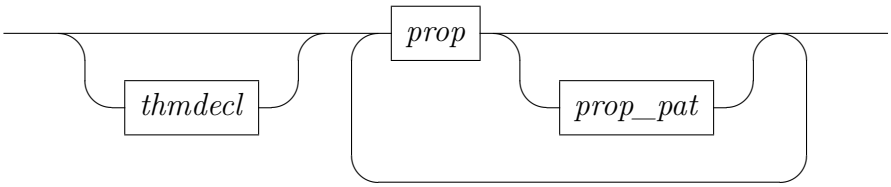


Declarations of local variables $x :: \tau$ and logical propositions $a : \varphi$ represent different views on the same principle of introducing a local scope. In practice, one may usually omit the typing of *vars* (due to type-inference), and the naming of propositions (due to implicit references of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

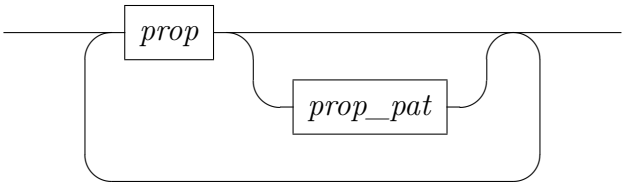
vars



props



props′

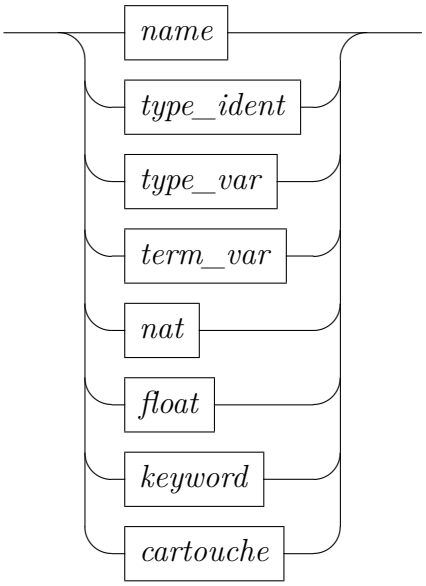


The treatment of multiple declarations corresponds to the complementary focus of *vars* versus *props*. In "$x_1 \ldots x_n :: \tau$" the typing refers to all variables, while in a: $\varphi_1 \ldots \varphi_n$ the naming refers to all propositions collectively. Isar language elements that refer to *vars*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

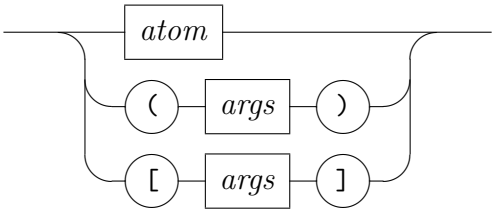
3.3.9 Attributes and theorems

Attributes have their own "semi-inner" syntax, in the sense that input conforming to *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 *atom* refers to any atomic entity, including any *keyword* conforming to *sym_ident*.

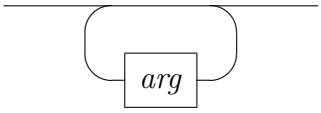
atom



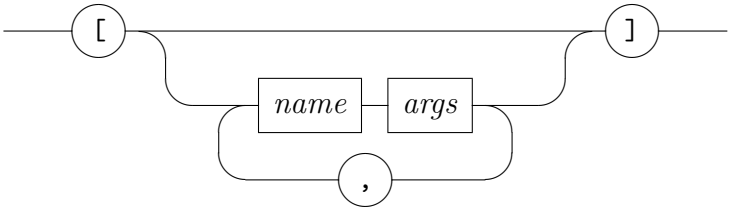
arg



args



attributes



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 *thm* and *thms*, the former requires an actual singleton result.

There are three forms of theorem references:

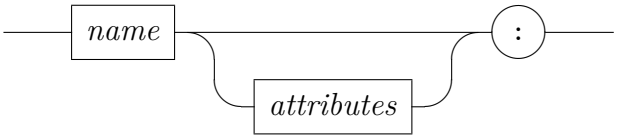
1. named facts a,
2. selections from named facts $a(i)$ or $a(j - k)$,
3. literal fact propositions using token syntax *altstring* ‘φ‘ or *cartouche* ‹φ› (see also method *fact*).

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.

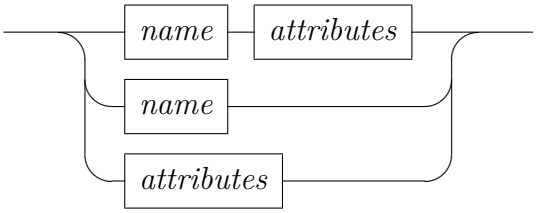
An extra pair of brackets around attributes (like “[[*simproc a*]]”) abbreviates a theorem reference involving an internal dummy fact, which will be ignored

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 **declare** and **using**.

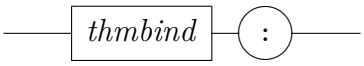
axmdecl



thmbind



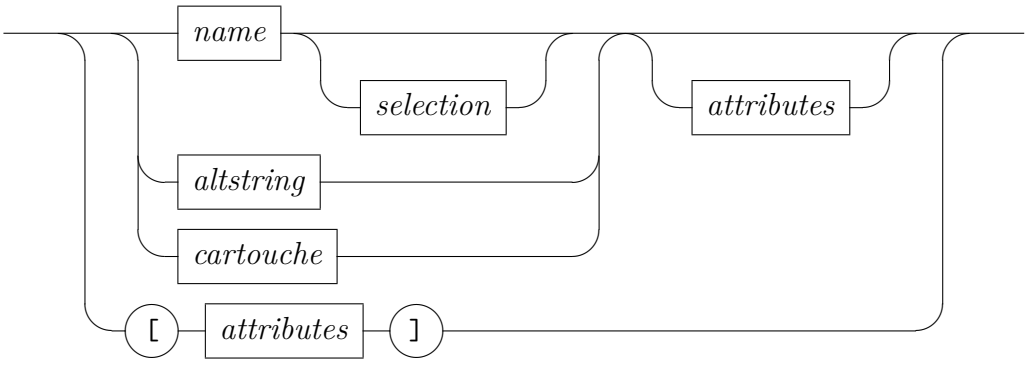
thmdecl



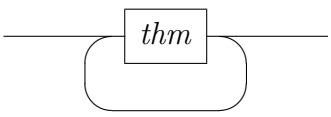
thmdef



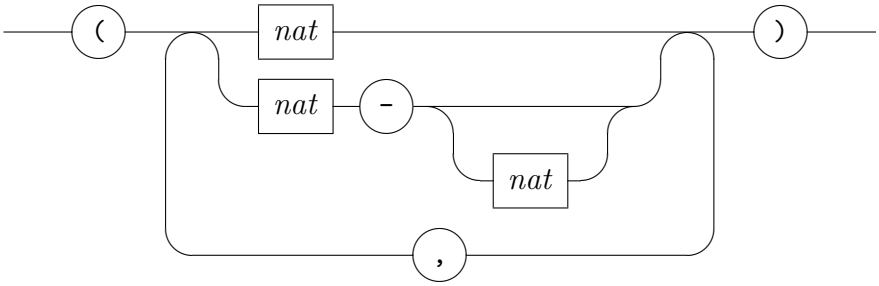
thm



thms



selection

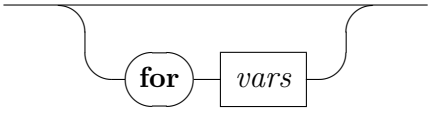


3.3.10 Structured specifications

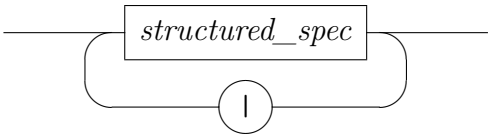
Structured specifications use propositions with explicit notation for the "eigen-context" to describe rule structure: $\bigwedge x.\ A\ x \Longrightarrow \ldots \Longrightarrow B\ x$ is expressed as $B\ x$ **if** $A\ x$ **and** ... **for** x. It is also possible to use dummy terms "_" (underscore) to refer to locally fixed variables anonymously.

Multiple specifications are delimited by "|" to emphasize separate cases: each with its own scope of inferred types for free variables.

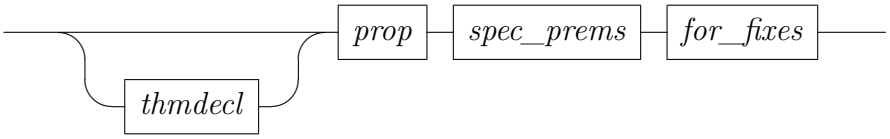
for_fixes



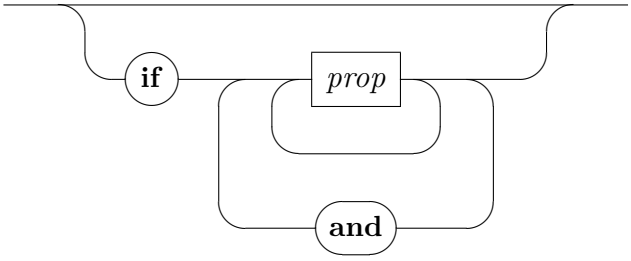
multi_specs



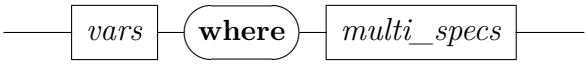
structured_spec



spec_prems

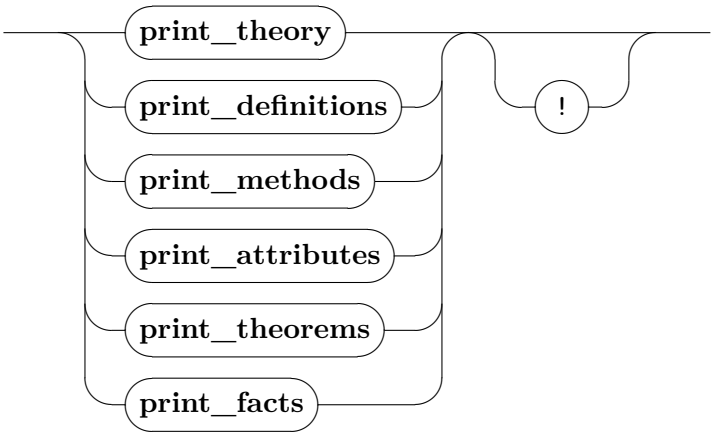


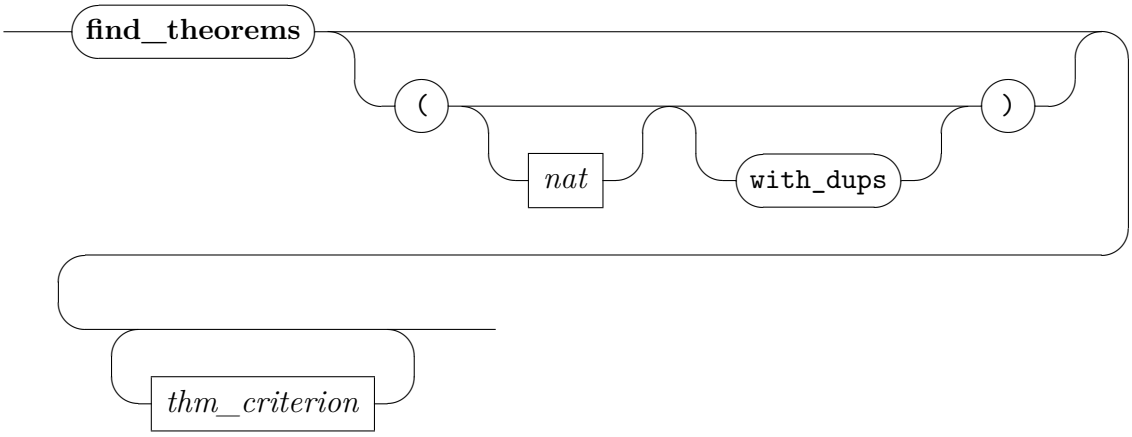
specification



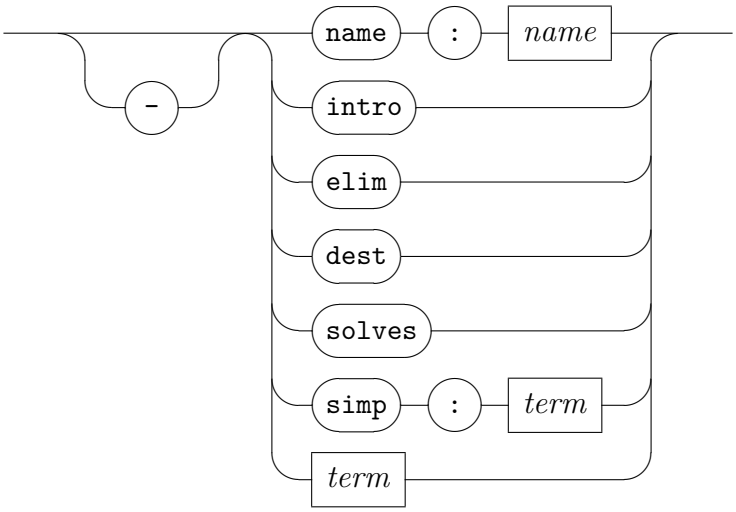
3.4 Diagnostic commands

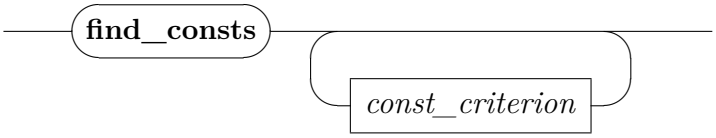
print_theory* : *context* →
print_definitions* : *context* →
print_methods* : *context* →
print_attributes* : *context* →
print_theorems* : *context* →
find_theorems* : *context* →
find_consts* : *context* →
thm_deps* : *context* →
unused_thms* : *context* →
print_facts* : *context* →
print_term_bindings* : *context* →



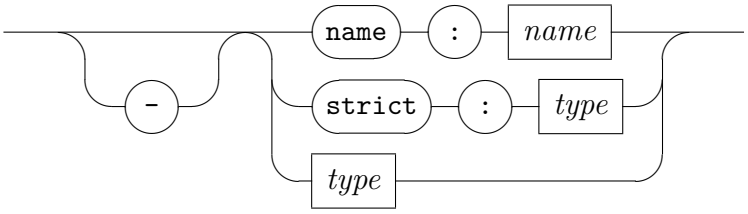


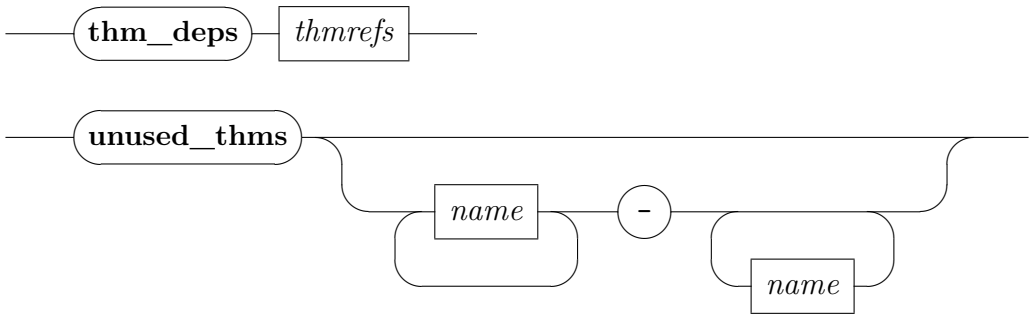
thm_criterion





const_criterion





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_theory prints the main logical content of the background theory; the "!" option indicates extra verbosity.

print_definitions prints dependencies of definitional specifications within the background theory, which may be constants (§5.4, §5.9) or types (§5.13.2, §11.6); the "!" option indicates extra verbosity.

print_methods prints all proof methods available in the current theory context; the "!" option indicates extra verbosity.

print_attributes prints all attributes available in the current theory context; the "!" option indicates extra verbosity.

print_theorems prints theorems of the background theory resulting from the last command; the "!" option indicates extra verbosity.

print_facts prints all local facts of the current context, both named and unnamed ones; the "!" option indicates extra verbosity.

print_term_bindings prints all term bindings that are present in the context.

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 *thms* prints immediate theorem dependencies, i.e. the union of all theorems that are used directly to prove the argument facts, without going deeper into the dependency graph.

unused_thms $A_1 \ \ldots \ A_m - B_1 \ \ldots \ B_n$ displays all theorems that are proved in theories $B_1 \ \ldots \ B_n$ or their parents but not in $A_1 \ \ldots \ A_m$ or their parents and that are never used. 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.

Chapter 4

Document preparation

Isabelle/Isar provides a simple document preparation system based on PDF-LaTeX, with support for hyperlinks and bookmarks within that format. This allows to produce papers, books, theses etc. from Isabelle theory sources.

LaTeX output is generated while processing a *session* in batch mode, as explained in the *The Isabelle System Manual* [54]. The main Isabelle tools to get started with document preparation are `isabelle mkroot` and `isabelle build`.

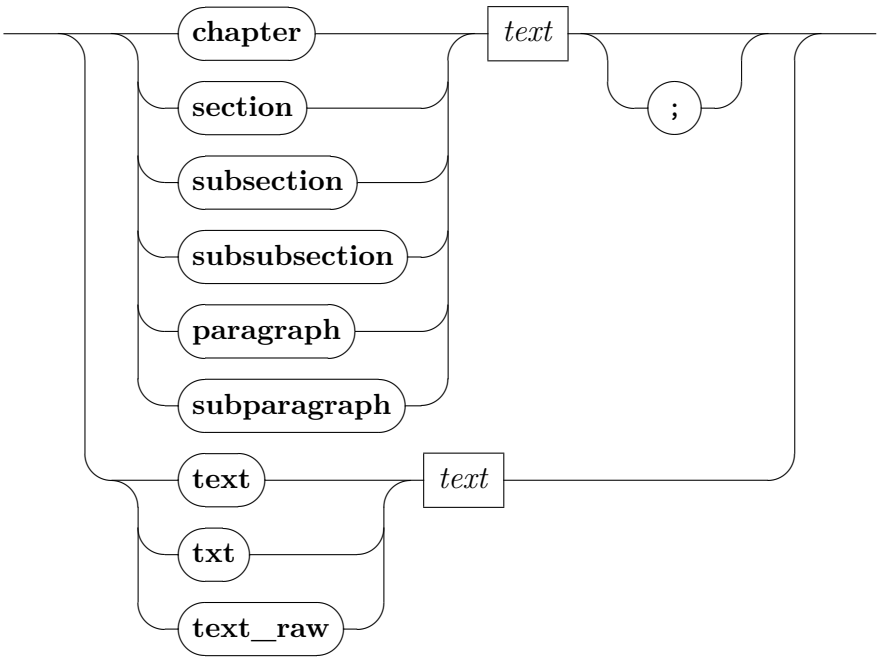
The classic Isabelle/HOL tutorial [38] also explains some aspects of theory presentation.

4.1 Markup commands

chapter	:	*any* → *any*
section	:	*any* → *any*
subsection	:	*any* → *any*
subsubsection	:	*any* → *any*
paragraph	:	*any* → *any*
subparagraph	:	*any* → *any*
text	:	*any* → *any*
txt	:	*any* → *any*
text_raw	:	*any* → *any*

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 (§3.3.5) are similar to markup commands, but have a different status within Isabelle/Isar syntax.



chapter, **section**, **subsection** etc. mark section headings within the theory source. This works in any context, even before the initial **theory** command. The corresponding LaTeX macros are `\isamarkupchapter`, `\isamarkupsection`, `\isamarkupsubsection` etc.

text and **txt** specify paragraphs of plain text. This corresponds to a LaTeX environment `\begin{isamarkuptext}` ... `\end{isamarkuptext}` etc.

text_raw is similar to **text**, but without any surrounding markup environment. This allows to inject arbitrary LaTeX source into the generated document.

All 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.

The proof markup commands closely resemble those for theory specifications, but have a different formal status and produce different LaTeX macros.

4.2 Document antiquotations

theory : *antiquotation*
thm : *antiquotation*
lemma : *antiquotation*
prop : *antiquotation*
term : *antiquotation*
term_type : *antiquotation*
typeof : *antiquotation*
const : *antiquotation*
abbrev : *antiquotation*
typ : *antiquotation*
type : *antiquotation*
class : *antiquotation*
locale : *antiquotation*
bundle : *antiquotation*
text : *antiquotation*
goals : *antiquotation*
subgoals : *antiquotation*
prf : *antiquotation*
full_prf : *antiquotation*
ML_text : *antiquotation*
ML : *antiquotation*
ML_def : *antiquotation*
ML_ref : *antiquotation*
ML_infix : *antiquotation*
ML_infix_def : *antiquotation*
ML_infix_ref : *antiquotation*
ML_type : *antiquotation*
ML_type_def : *antiquotation*
ML_type_ref : *antiquotation*
ML_structure : *antiquotation*
ML_structure_def : *antiquotation*
ML_structure_ref : *antiquotation*
ML_functor : *antiquotation*
ML_functor_def : *antiquotation*
ML_functor_ref : *antiquotation*

emph	:	*antiquotation*
bold	:	*antiquotation*
verbatim	:	*antiquotation*
bash_function	:	*antiquotation*
system_option	:	*antiquotation*
session	:	*antiquotation*
file	:	*antiquotation*
url	:	*antiquotation*
cite	:	*antiquotation*
nocite	:	*antiquotation*
citet	:	*antiquotation*
citep	:	*antiquotation*
print_antiquotations*	:	*context* →

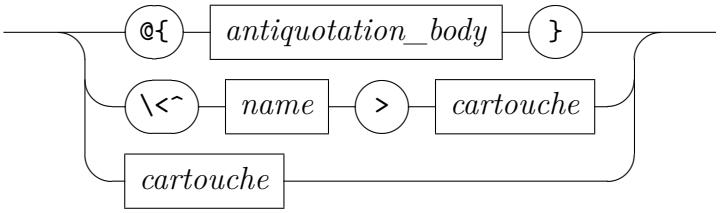
The overall content of an Isabelle/Isar theory may alternate between formal and informal text. The main body consists of formal specification and proof commands, interspersed with markup commands (§4.1) or document comments (§3.3.5). The argument of markup commands quotes informal text to be printed in the resulting document, but may again refer to formal entities via *document antiquotations*.

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

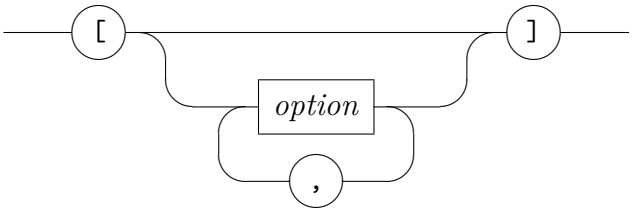
Antiquotations usually spare the author tedious typing of logical entities in full detail. Even more importantly, some degree of consistency-checking between the main body of formal text and its informal explanation is achieved, since terms and types appearing in antiquotations are checked within the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

Antiquotations are in general written as `@{`*name* `[`*options*`]` *arguments*`}`. The short form `\<^`*name*`>`‹*argument_content*› (without surrounding `@{...}`) works for a single argument that is a cartouche. A cartouche without special decoration is equivalent to `\<^cartouche>`‹*argument_content*›, which is equivalent to `@{cartouche` ‹*argument_content*›`}`. The special name *cartouche* is defined in the context: Isabelle/Pure introduces that as an alias to *text* (see below). Consequently, ‹*foo_bar* + *baz* ≤ *bazar*› prints literal quasi-formal text (unchecked). A control symbol `\<^`*name*`>` within the body text, but without a subsequent cartouche, is equivalent to `@{`*name*`}`.

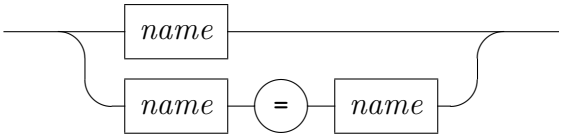
antiquotation



options

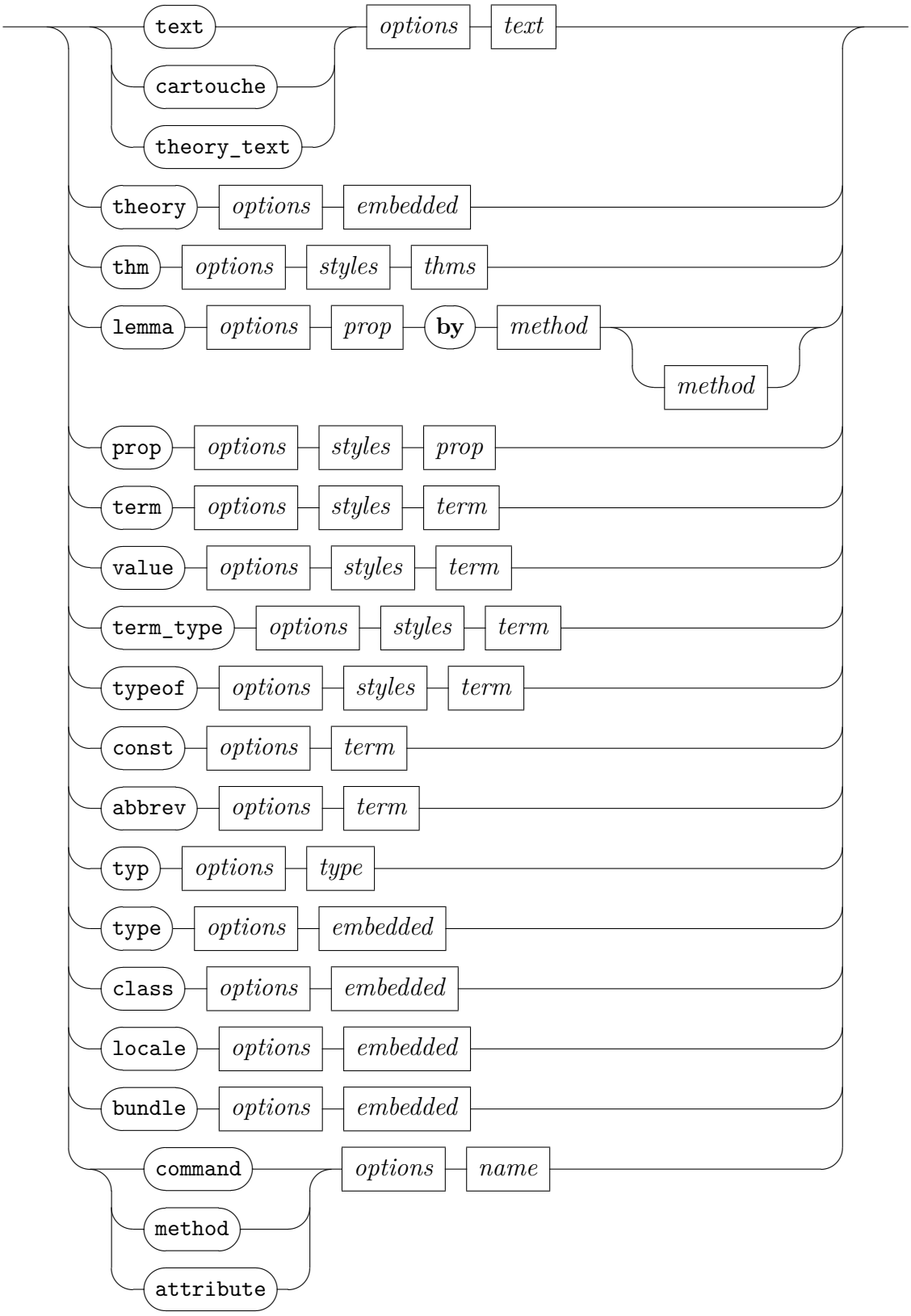


option

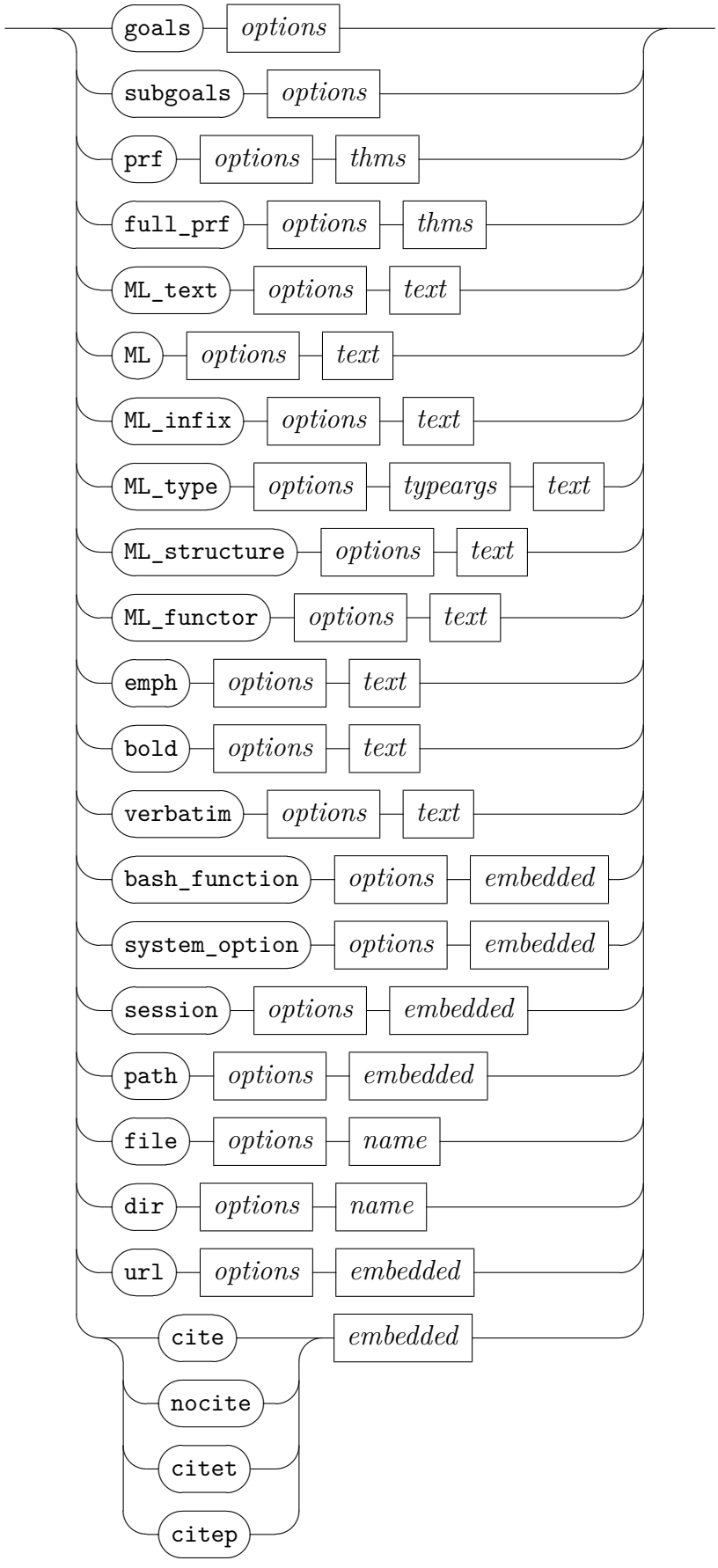


Note that the syntax of antiquotations may *not* include source comments `(* ... *)` nor verbatim text `{* ... *}`.

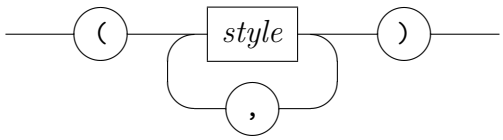
antiquotation_body



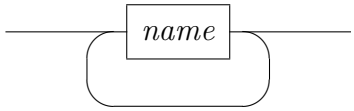
antiquotation

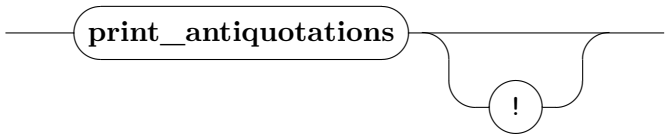


styles



style





@{*text s*} prints uninterpreted source text *s*, i.e. inner syntax. 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.

It is also possible to write this in the short form ‹*s*› without any further decoration.

@{*theory_text s*} prints uninterpreted theory source text *s*, i.e. outer syntax with command keywords and other tokens.

@{*theory A*} prints the session-qualified theory name *A*, which is guaranteed to refer to a valid ancestor theory in the current context.

@{*thm* a_1 ... a_n} prints theorems a_1 ... a_n. Full fact expressions are allowed here, including attributes (§3.3.9).

@{*prop* φ} prints a well-typed proposition φ.

@{*lemma* φ *by m*} proves a well-typed proposition φ by method *m* and prints the original φ.

@{*term t*} prints a well-typed term *t*.

@{*value t*} evaluates a term *t* and prints its result, see also **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 ... x_n} prints a constant abbreviation $c\ x_1 \ldots x_n \equiv rhs$ as defined in the current context.

@{*typ* τ} prints a well-formed type τ.

@{*type* κ} prints a (logical or syntactic) type constructor κ.

@{*class c*} prints a class *c*.

@{*locale c*} prints a locale *c*.

@{*bundle c*} prints a bundle *c*.

@{*command name*}, @{*method name*}, @{*attribute name*} print checked entities of the Isar language.

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

@{*ML_text s*} prints ML text verbatim: only the token language is checked.

@{*ML s*}, @{*ML_infix s*}, @{*ML_type s*}, @{*ML_structure s*}, and @{*ML_functor s*} check text *s* as ML value, infix operator, type, exception, structure, and functor respectively. The source is printed verbatim. The variants @{*ML_def s*} and @{*ML_ref s*} etc. maintain the document index: "def" means to make a bold entry, "ref" means to make a regular entry.

There are two forms for type constructors, with or without separate type arguments: this impacts only the index entry. For example, @{*ML_type_ref* ‹*'a list*›} makes an entry literally for "*'a list*" (sorted under the letter "a"), but @{*ML_type_ref 'a* ‹*list*›} makes an entry for the constructor name "*list*".

@{*emph s*} prints document source recursively, with LaTeX markup `\emph{...}`.

@{*bold s*} prints document source recursively, with LaTeX markup `\textbf{...}`.

@{*verbatim s*} prints uninterpreted source text literally as ASCII characters, using some type-writer font style.

@{*bash_function name*} prints the given GNU bash function verbatim. The name is checked wrt. the Isabelle system environment [54].

@{*system_option name*} prints the given system option verbatim. The name is checked wrt. cumulative `etc/options` of all Isabelle components, notably `~~/etc/options`.

@{*session name*} prints given session name verbatim. The name is checked wrt. the dependencies of the current session.

@{*path name*} prints the file-system path name verbatim.

@{*file name*} is like @{*path name*}, but ensures that *name* refers to a plain file.

@{*dir name*} is like @{*path name*}, but ensures that *name* refers to a directory.

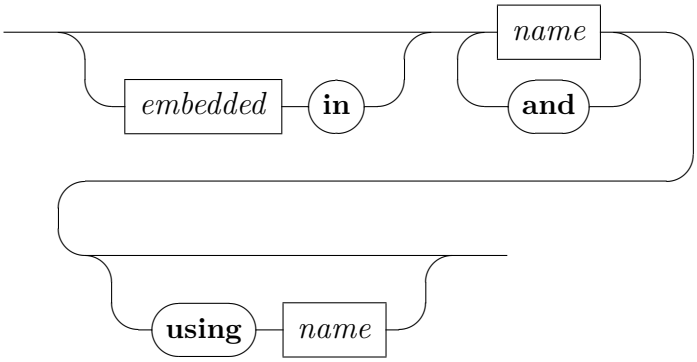
@{*url name*} produces markup for the given URL, which results in an active hyperlink within the text.

@{*cite arg*} produces the BibTeX citation macro `\cite[...]{...}` with its optional and mandatory argument. The analogous `\nocite`, and the `\citet` and `\citep` variants from the `natbib` package[1] are supported as well.

The argument syntax is uniform for all variants and is usually presented in control-symbol-cartouche form: ***cite*** ‹*arg*›. The formal syntax of the nested argument language is defined as follows:

[1] https://ctan.org/pkg/natbib

arg



Here the embedded text is free-form LaTeX, which becomes the optional argument of the `\cite` macro. The named items are BibTeX database entries and become the mandatory argument (separated by comma). The optional part "**using** *name*" specifies an alternative LaTeX macro name.

print_antiquotations prints all document antiquotations that are defined in the current context; the "!" option indicates extra verbosity.

4.2.1 Styled antiquotations

The antiquotations *thm*, *prop* and *term* admit an 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:

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.

rhs is like *lhs*, but extracts the second argument.

concl extracts the conclusion C from a rule in Horn-clause normal form $A_1 \Longrightarrow \ldots\ A_n \Longrightarrow C$.

prem n extract premise number n from from a rule in Horn-clause normal form $A_1 \Longrightarrow \ldots\ A_n \Longrightarrow C$.

4.2.2 General options

The following options are available to tune the printed output of antiquotations. Note that many of these coincide with system and configuration options of the same names.

show_types = *bool* and *show_sorts* = *bool* control printing of explicit type and sort constraints.

show_structs = *bool* controls printing of implicit structures.

show_abbrevs = *bool* controls folding of abbreviations.

names_long = *bool* forces names of types and constants etc. to be printed in their fully qualified internal form.

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.

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 *false* for more concise output.

eta_contract = *bool* prints terms in η-contracted form.

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.

break = *bool* controls line breaks in non-display material.

cartouche = *bool* indicates if the output should be delimited as cartouche.

quotes = *bool* indicates if the output should be delimited via double quotes (option *cartouche* takes precedence). Note that the Isabelle LaTeX styles may suppress quotes on their own account.

mode = *name* adds *name* to the print mode to be used for presentation. Note that the standard setup for LaTeX output is already present by default, with mode "*latex*".

margin = *nat* and *indent* = *nat* change the margin or indentation for pretty printing of display material.

goals_limit = *nat* determines the maximum number of subgoals 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 *thm*, *term*, etc. is still enabled; use the *text* antiquotation for unchecked output.

Regular *term* and *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.

Cartouche delimiters of the argument are stripped for antiquotations that are internally categorized as "embedded".

source_cartouche is like *source*, but cartouche delimiters are always preserved in the output.

For Boolean flags, "*name* = *true*" may be abbreviated as "*name*". All of the above flags are disabled by default, unless changed specifically for a logic session in the corresponding `ROOT` file.

4.3 Markdown-like text structure

The markup commands **text**, **txt**, **text_raw** (§4.1) consist of plain text. Its internal structure consists of paragraphs and (nested) lists, using special Isabelle symbols and some rules for indentation and blank lines. This quasi-visual format resembles *Markdown*[2], but the full complexity of that notation is avoided.

This is a summary of the main principles of minimal Markdown in Isabelle:

- List items start with the following markers

[2]http://commonmark.org

itemize: `\<^item>`

enumerate: `\<^enum>`

description: `\<^descr>`

- Adjacent list items with same indentation and same marker are grouped into a single list.
- Singleton blank lines separate paragraphs.
- Multiple blank lines escape from the current list hierarchy.

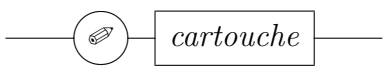
Notable differences to official Markdown:

- Indentation of list items needs to match exactly.
- Indentation is unlimited (official Markdown interprets four spaces as block quote).
- List items always consist of paragraphs — there is no notion of "tight" list.
- Section headings are expressed via Isar document markup commands (§4.1).
- URLs, font styles, other special content is expressed via antiquotations (§4.2), usually with proper nesting of sub-languages via text cartouches.

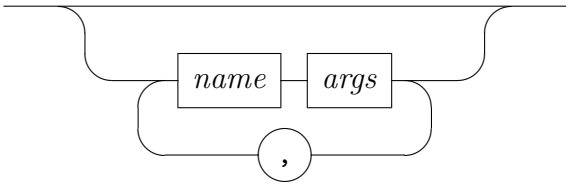
4.4 Document markers and command tags

Document markers are formal comments of the form ✎‹*marker_body*› (using the control symbol `\<^marker>`) and may occur anywhere within the outer syntax of a command: the inner syntax of a marker body resembles that for attributes (§3.3.9). In contrast, *Command tags* may only occur after a command keyword and are treated as special markers as explained below.

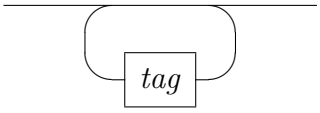
marker



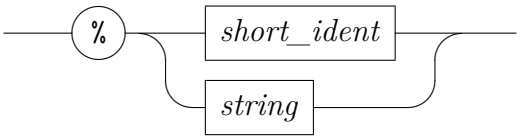
marker_body



tags

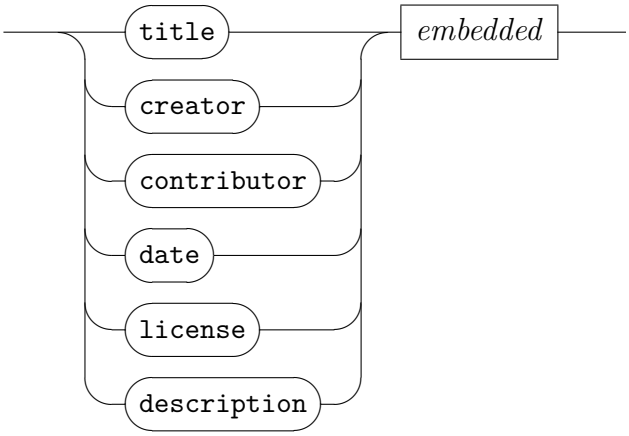


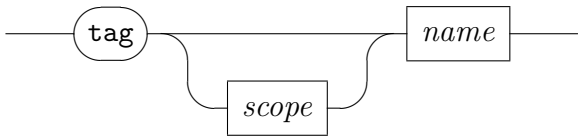
tag



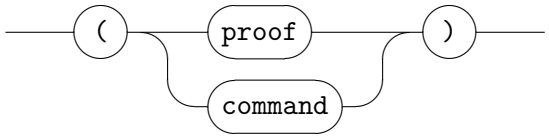
Document markers are stripped from the document output, but surrounding white-space is preserved: e.g. a marker at the end of a line does not affect the subsequent line break. Markers operate within the semantic presentation context of a command, and may modify it to change the overall appearance of a command span (e.g. by adding tags).

Each document marker has its own syntax defined in the theory context; the following markers are predefined in Isabelle/Pure:





scope



✎‹*title arg*›, ✎‹*creator arg*›, ✎‹*contributor arg*›, ✎‹*date arg*›, ✎‹*license arg*›, and ✎‹*description arg*› produce markup in the PIDE document, without any immediate effect on typesetting. This vocabulary is taken from the Dublin Core Metadata Initiative[3]. The argument is an uninterpreted string, except for *description*, which consists of words that are subject to spell-checking.

✎‹*tag name*› updates the list of command tags in the presentation context: later declarations take precedence, so ✎‹*tag a*, *tag b*, *tag c*› produces a reversed list. The default tags are given by the original **keywords** declaration of a command, and the system option `document_tags`.

The optional *scope* tells how far the tagging is applied to subsequent proof structure: "(*proof*)" means it applies to the following proof text, and "(*command*)" means it only applies to the current command. The default within a proof body is "(*proof*)", but for toplevel goal statements it is "(*command*)". Thus a *tag* marker for **theorem**, **lemma** etc. does *not* affect its proof by default.

An old-style command tag `%`*name* is treated like a document marker ✎‹*tag* (*proof*) *name*›: the list of command tags precedes the list of document markers. The head of the resulting tags in the presentation context is turned into LaTeX environments to modify the type-setting. The following tags are pre-declared for certain classes of commands, and serve as default markup for certain kinds of commands:

document	document markup commands
theory	theory begin/end
proof	all proof commands
ML	all commands involving ML code

[3]https://www.dublincore.org/specifications/dublin-core/dcmi-terms

The Isabelle document preparation system [54] allows tagged command regions to be presented specifically, e.g. to fold proof texts, or drop parts of the text completely.

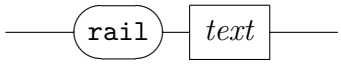
For example "**by** *auto* ✎‹*tag invisible*›" causes that piece of proof to be treated as *invisible* instead of *proof* (the default), which may be shown or hidden depending on the document setup. In contrast, "**by** *auto* ✎‹*tag visible*›" forces this text to be shown invariably.

Explicit tag specifications within a proof apply to all subsequent commands of the same level of nesting. For example, "**proof** ✎‹*tag invisible*› ... **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 [54] for further details.

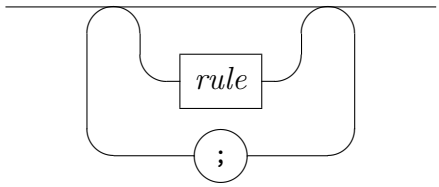
4.5 Railroad diagrams

rail : *antiquotation*

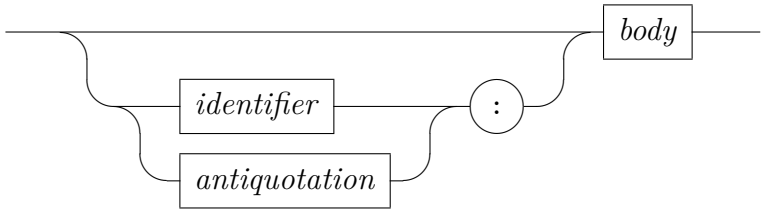


The *rail* antiquotation allows to include syntax diagrams into Isabelle documents. LaTeX requires the style file `~~/lib/texinputs/railsetup.sty`, which can be used via `\usepackage{railsetup}` in `root.tex`, for example.

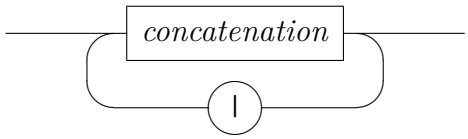
The rail specification language is quoted here as Isabelle *string* or text *cartouche*; it has its own grammar given below.



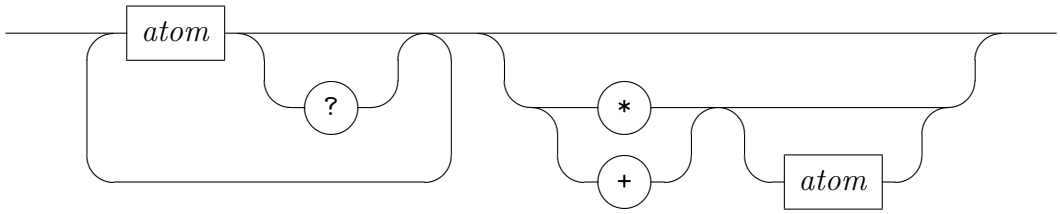
rule



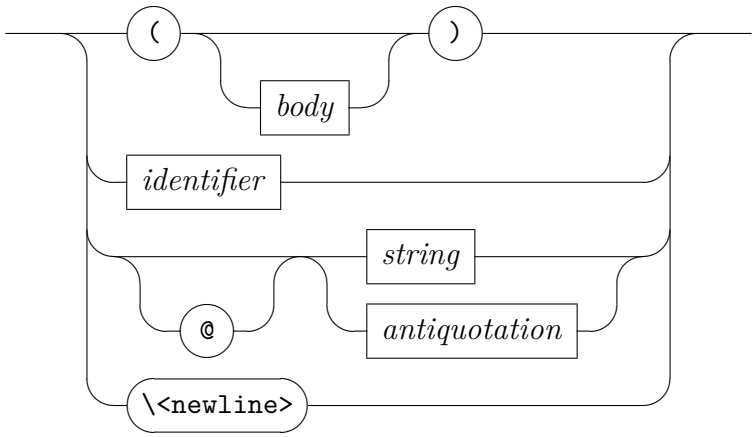
body



concatenation



atom



The lexical syntax of *identifier* coincides with that of *short_ident* in regular Isabelle syntax, but *string* uses single quotes instead of double quotes of the standard *string* category.

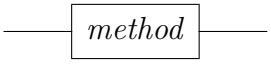
Each *rule* defines a formal language (with optional name), using a notation that is similar to EBNF or regular expressions with recursion. The meaning and visual appearance of these rail language elements is illustrated by the following representative examples.

- Empty `()`

- Nonterminal `A`



- Nonterminal via Isabelle antiquotation `@{syntax method}`



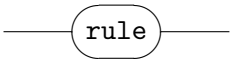
- Terminal `'xyz'`



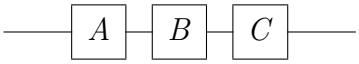
- Terminal in keyword style `@'xyz'`



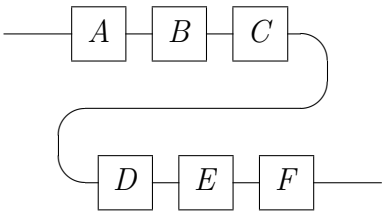
- Terminal via Isabelle antiquotation `@@{method rule}`



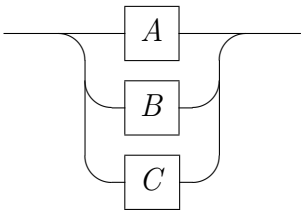
- Concatenation `A B C`



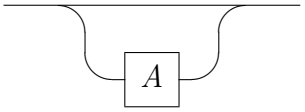
- Newline inside concatenation `A B C \<newline> D E F`



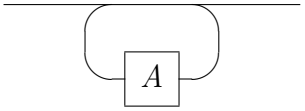
- Variants `A | B | C`



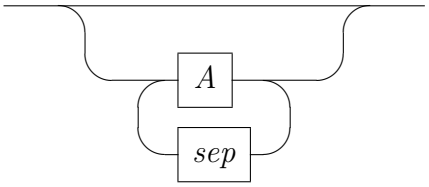
- Option `A ?`



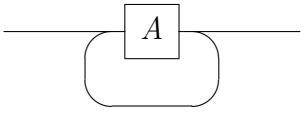
- Repetition `A *`



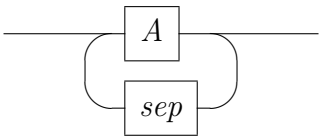
- Repetition with separator `A * sep`



- Strict repetition `A +`



- Strict repetition with separator `A + sep`


A
sep

Chapter 5

Specifications

The Isabelle/Isar theory format integrates specifications and proofs, with support for interactive development by continuous document editing. There is a separate 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**).

5.1 Defining theories

theory	:	*toplevel* $\to$ *theory*
end	:	*theory* $\to$ *toplevel*
thy_deps*	:	*theory* $\to$

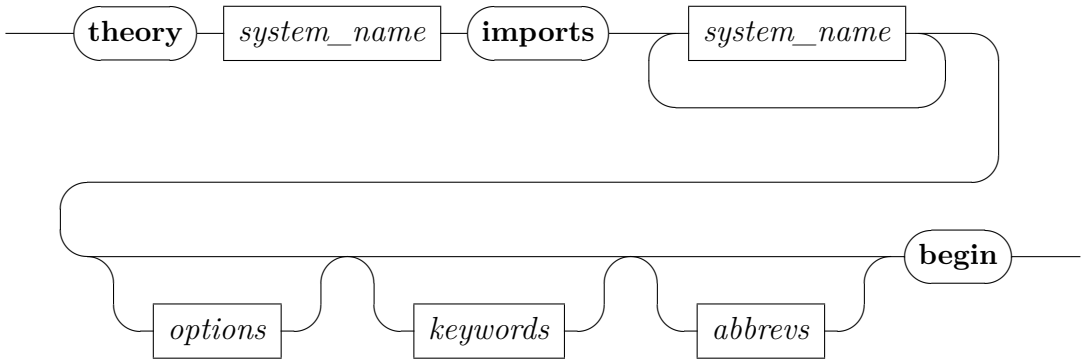
Isabelle/Isar theories are defined via theory files, which consist of an outermost sequence of definition–statement–proof elements. Some definitions are self-sufficient (e.g. **fun** in Isabelle/HOL), with foundational proofs performed internally. Other definitions require an explicit proof as justification (e.g. **function** and **termination** in Isabelle/HOL). Plain statements like **theorem** or **lemma** are merely a special case of that, defining a theorem from a given proposition and its proof.

The theory body may be sub-structured by means of *local theory targets*, such as **locale** and **class**. It is also possible to use **context begin** ... **end** blocks to delimited a local theory context: a *named context* to augment a locale or class specification, or an *unnamed context* to refer to local parameters and assumptions that are discharged later. See §5.2 for more details.

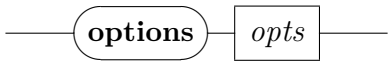
A theory is commenced by the **theory** command, which indicates imports of previous theories, according to an acyclic foundational order. Before the initial **theory** command, there may be optional document header material

(like **section** or **text**, see §4.1). The document header is outside of the formal theory context, though.

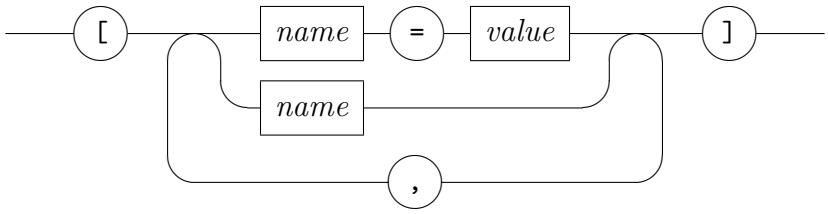
A theory is concluded by a final **end** command, one that does not belong to a local theory target. No further commands may follow such a global **end**.



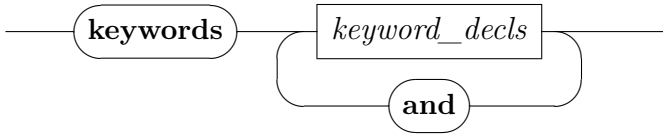
options



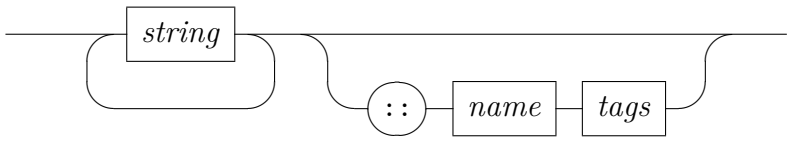
opts



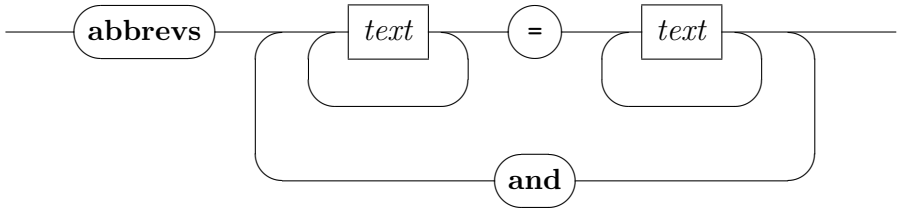
keywords

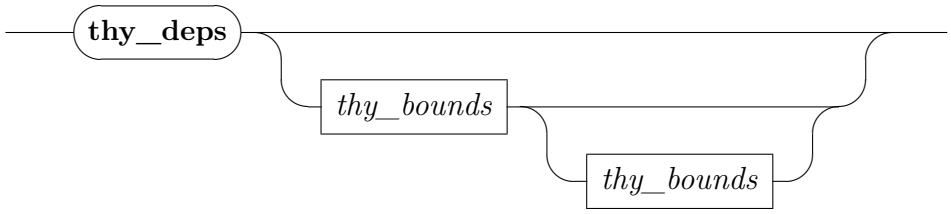


keyword_decls

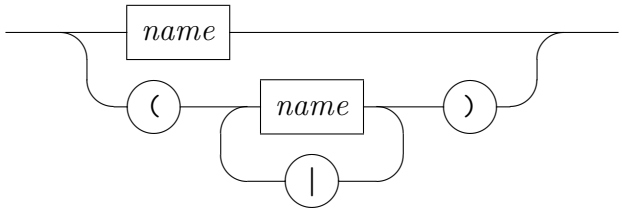


abbrevs





thy_bounds



theory *A* **imports** $B_1 \ldots B_n$ **begin** starts a new theory *A* based on the merge of existing theories $B_1 \ldots B_n$. Due to the possibility to import more than one ancestor, the resulting theory structure of an Isabelle session forms a directed acyclic graph (DAG). Isabelle takes care that sources contributing to the development graph are always up-to-date: changed files are automatically rechecked whenever a theory header specification is processed.

Empty imports are only allowed in the bootstrap process of the special theory *Pure*, which is the start of any other formal development based on Isabelle. Regular user theories usually refer to some more complex entry point, such as theory *Main* in Isabelle/HOL.

The **options** specification declares options that are only used for the subsequent theory body — neither for the theory imports nor for other theories importing the current theory. This is analogous to session options within a `ROOT` file [54], but the scope of options is different, and inlined theory options may be edited within a running PIDE session.

The **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_defn* or **keywords** `"datatype"` :: *thy_defn* for theory-level definitions with and without proof, respectively. Additional *tags* provide defaults for document preparation (§4.4).

The **abbrevs** specification declares additional abbreviations for syntactic completion. The default for a new keyword is just its name, but completion may be avoided by defining **abbrevs** with empty text.

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.

thy_deps visualizes the theory hierarchy as a directed acyclic graph. By default, all imported theories are shown. This may be restricted by specifying bounds wrt. the theory inclusion relation.

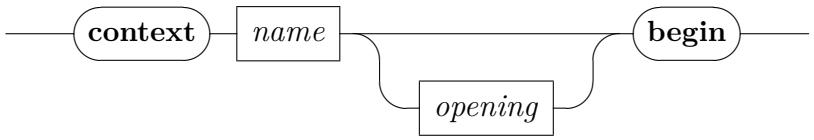
5.2 Local theory targets

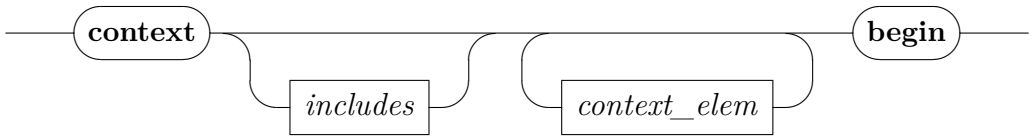
context : *theory* → *local_theory*
end : *local_theory* → *theory*
private
qualified

A local theory target is a specification context that is 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.

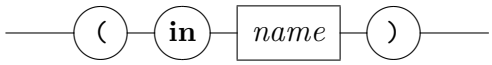
Named contexts refer to locales (cf. §5.7) or type classes (cf. §5.8); the name "−" signifies the global theory context.

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**.





target



context *c bundles* **begin** opens a named context, by recommencing an existing locale or class *c*. Note that locale and class definitions allow to include the **begin** keyword as well, in order to continue the local theory immediately after the initial specification. Optionally given *bundles* only take effect in the surface context within the **begin** / **end** block.

context *bundles elements* **begin** opens an unnamed context, by extending the enclosing global or local theory target by the given declaration bundles (§5.3) and context elements (**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 concludes the current local theory, according to the nesting of contexts. Note that a global **end** has a different meaning: it concludes the theory itself (§5.1).

private or **qualified** may be given as modifiers before any local theory command. This restricts name space accesses to the local scope, as determined by the enclosing **context begin** ... **end** block. Outside its scope, a **private** name is inaccessible, and a **qualified** name is only accessible with some qualification.

Neither a global **theory** nor a **locale** target provides a local scope by itself: an extra unnamed context is required to use **private** or **qualified** here.

(**in** *c*) given after any local theory command specifies an immediate target, e.g. "**definition** (**in** *c*)" or "**theorem** (**in** *c*)". This works both in

a local or global theory context; the current target context will be suspended for this command only. Note that "(**in** −)" will always produce a global result independently of the current target context.

Any specification element that operates on *local_theory* according to this manual implicitly allows the above target syntax (**in** *c*), but individual syntax diagrams omit that aspect for clarity.

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

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

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

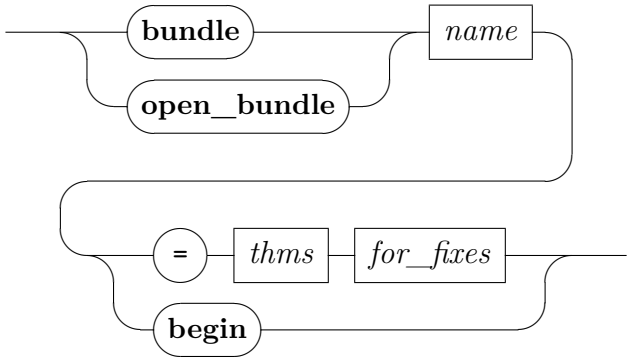
5.3 Bundled declarations

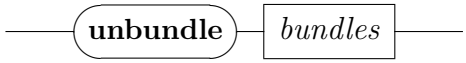
bundle	:	*local_theory* → *local_theory*
bundle	:	*theory* → *local_theory*
unbundle	:	*local_theory* → *local_theory*
print_bundles*	:	*context* →
include	:	*proof*(*state*) → *proof*(*state*)
including	:	*proof*(*prove*) → *proof*(*prove*)
includes	:	*syntax*
opening	:	*syntax*

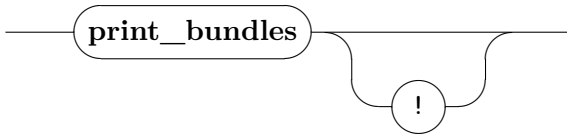
The outer syntax of fact expressions (§3.3.9) involves theorems and attributes, which are evaluated in the context and applied to it. Attributes may declare theorems to the context, as in *this_rule* [*intro*] *that_rule* [*elim*] for example. Configuration options (§9.1) are special declaration attributes that operate on the context without a theorem, as in [[*show_types* = *false*]] for example.

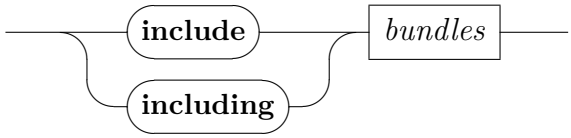
Expressions of this form may be defined as *bundled declarations* in the context, and included in other situations later on. Including declaration bundles

augments a local context casually without logical dependencies, which is in contrast to locales and locale interpretation (§5.7).

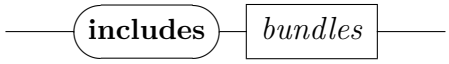




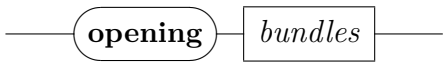




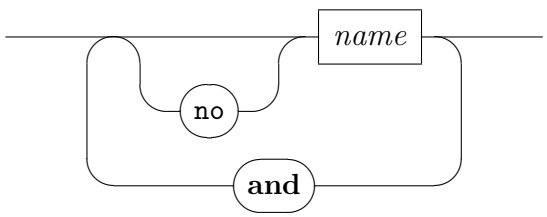
includes



opening



bundles



bundle $b = \mathit{decls}$ 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.

bundle b **begin** *body* **end** defines a bundle of declarations from the *body* of local theory specifications. It may consist of commands that are technically equivalent to **declare** or **declaration**, which also includes **notation**, for example. Named fact declarations like "**lemmas** a [*simp*] $= b$" or "**lemma** a [*simp*]: B $\langle \mathit{proof} \rangle$" are also admitted, but the name bindings are not recorded in the bundle.

open_bundle b is like **bundle** b followed by **unbundle** b, so its declarations are activated immediately, but also named for later re-use.

unbundle $\overline{b}$ activates the declarations from the given bundles in the current local theory context. This is analogous to **lemmas** (§5.14) with the expanded bundles.

print_bundles prints the named bundles that are available in the current context; the "!" option indicates extra verbosity.

include $\overline{b}$ activates the declarations from the given bundles in a proof body (forward mode). This is analogous to **note** (§6.2.3) with the expanded bundles.

including $\overline{b}$ is similar to **include**, but works in proof refinement (backward mode). This is analogous to **using** (§6.2.3) with the expanded bundles.

includes $\overline{b}$ is similar to **include**, but applies to a confined specification context: unnamed **context**s and long statements of **theorem**.

opening $\overline{b}$ is similar to **includes**, but applies to a named specification context: **locale**s, **class**es and named **context**s. The effect is confined to the surface context within the specification block itself and the corresponding **begin** / **end** block.

Bundle names may be prefixed by the reserved word `no` to indicate that the polarity of certain declaration commands should be inverted, notably:

- **syntax** versus **no_syntax**
- **translations** versus **no_translations**

- **notation** versus **no_notation**
- **type_notation** versus **no_type_notation**
- **adhoc_overloading** versus **no_adhoc_overloading**

This also works recursively for the **unbundle** command as declaration inside a **bundle** definition: `no` means that both the order and polarity of declarations is reversed (following algebraic group laws).

Here is an artificial example of bundling various configuration options:

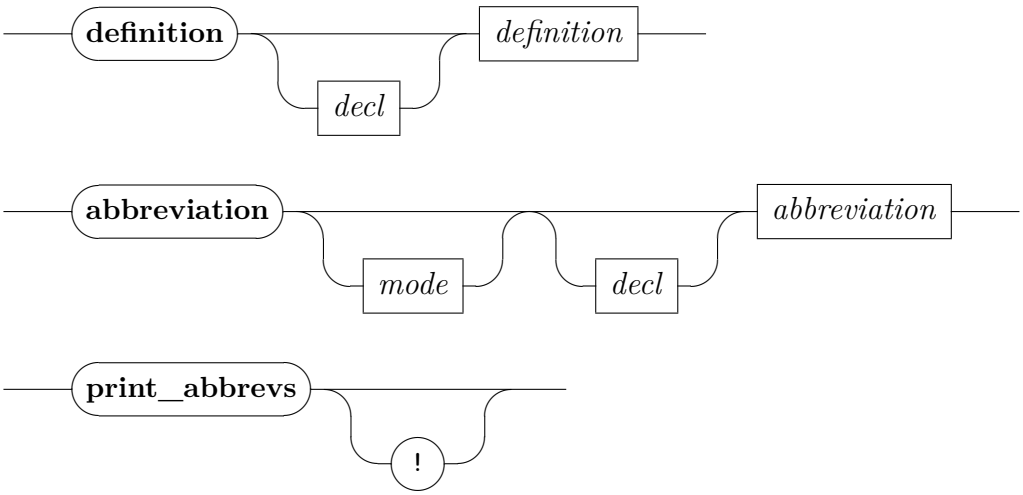
bundle *trace* = [[*simp_trace*, *linarith_trace*, *metis_trace*, *smt_trace*]]

lemma $x = x$
including *trace* **by** *metis*

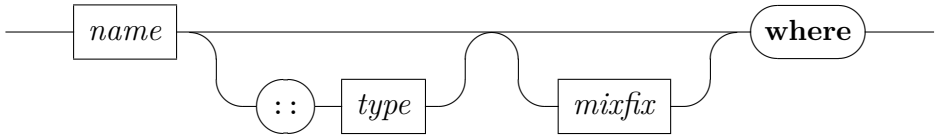
5.4 Term definitions

definition	:	*local_theory* → *local_theory*
defn	:	*attribute*
print_defn_rules*	:	*context* →
abbreviation	:	*local_theory* → *local_theory*
print_abbrevs*	:	*context* →

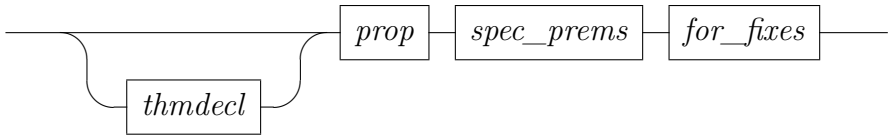
Term definitions may either happen within the logic (as equational axioms of a certain form (see also §5.9), or outside of it as rewrite system on abstract syntax. The second form is called "abbreviation".



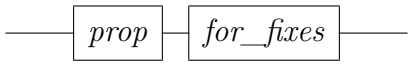
decl



definition



abbreviation



definition *c* **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 *defn* by the object-logic. This usually covers object-level equality $x = y$ and equivalence $A \longleftrightarrow B$. End-users normally need not change the *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 $y \neq 0 \Longrightarrow g\ x\ y = u$ instead of an unrestricted $g \equiv \lambda x\ y.\ u$.

print_defn_rules prints the definitional rewrite rules declared via *defn* in the current context.

abbreviation *c* **where** *eq* introduces a syntactic constant which is associated with a certain term according to the meta-level equality *eq*.

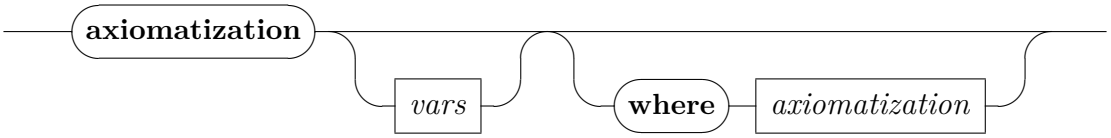
Abbreviations participate in the usual type-inference process, but are expanded before the logic ever sees them. Pretty printing of terms involves higher-order rewriting with rules stemming from reverted abbreviations. This needs some care to avoid overlapping or looping syntactic replacements!

The optional *mode* specification restricts output to a particular print mode; using "*input*" here achieves the effect of one-way abbreviations. The mode may also include an "**output**" qualifier that affects the concrete syntax declared for abbreviations, cf. **syntax** in §8.5.2.

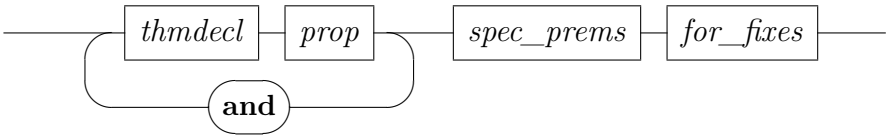
print_abbrevs prints all constant abbreviations of the current context; the "!" option indicates extra verbosity.

5.5 Axiomatizations

axiomatization : *theory* → *theory* (*axiomatic*!)



axiomatization



axiomatization $c_1 \ldots c_m$ **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 for the same constants later on, but it is always possible to emit axiomatizations without referring to particular constants. Note that lack of precise dependency tracking of axiomatizations may disrupt the well-formedness of an otherwise definitional theory.

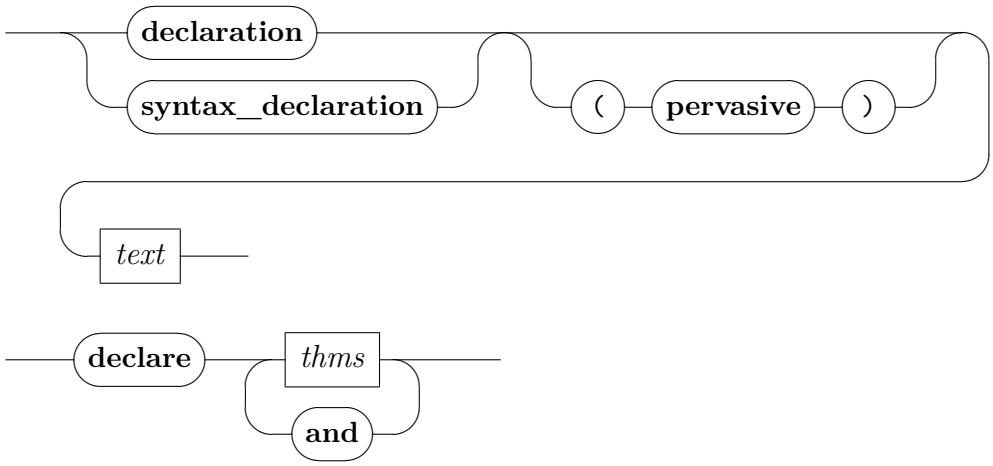
Axiomatization is restricted to a global theory context: support for local theory targets §5.2 would introduce an extra dimension of uncertainty what the written specifications really are, and make it infeasible to argue why they are correct.

Axiomatic specifications are required when declaring a new logical system within Isabelle/Pure, but in an application environment like Isabelle/HOL the user normally stays within definitional mechanisms provided by the logic and its libraries.

5.6 Generic declarations

declaration : *local_theory* → *local_theory*
syntax_declaration : *local_theory* → *local_theory*
declare : *local_theory* → *local_theory*

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.



declaration *d* adds the declaration function *d* of ML type `Morphism.declaration`, 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.

syntax_declaration is similar to **declaration**, but is meant to affect only "syntactic" tools by convention (such as notation and type-checking information).

declare *thms* declares theorems to the current local theory context. No theorem binding is involved here, unlike **lemmas** (cf. §5.14), so **declare**

only has the effect of applying attributes as included in the theorem specification.

5.7 Locales

A locale is a functor that maps parameters (including implicit type parameters) and a specification to a list of declarations. The syntax of locales is modeled after the Isar proof context commands (cf. §6.2.1).

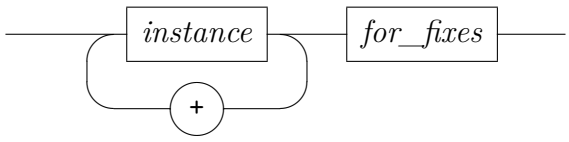
Locale hierarchies are supported by maintaining a graph of dependencies between locale instances in the global theory. Dependencies may be introduced through import (where a locale is defined as sublocale of the imported instances) or by proving that an existing locale is a sublocale of one or several locale instances.

A locale may be opened with the purpose of appending to its list of declarations (cf. §5.2). When opening a locale declarations from all dependencies are collected and are presented as a local theory. In this process, which is called *roundup*, redundant locale instances are omitted. A locale instance is redundant if it is subsumed by an instance encountered earlier. A more detailed description of this process is available elsewhere [4].

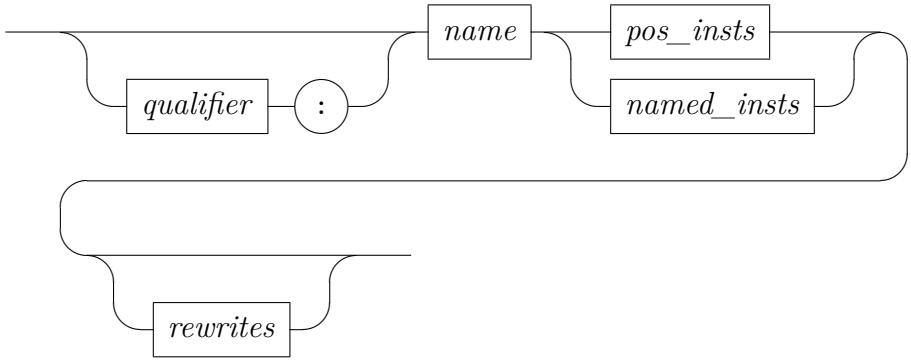
5.7.1 Locale expressions

A *locale expression* denotes a context composed of instances of existing locales. The context consists of the declaration elements from the locale instances. Redundant locale instances are omitted according to roundup.

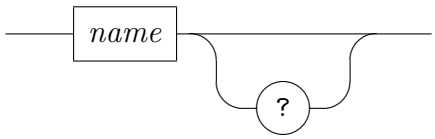
locale_expr



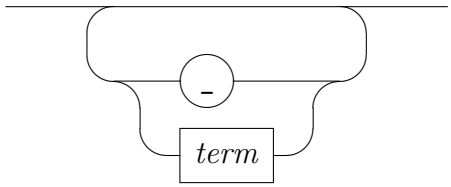
instance



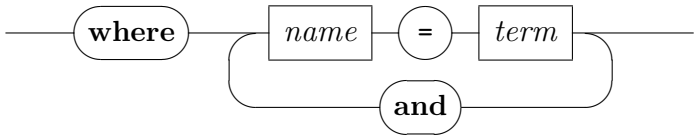
qualifier



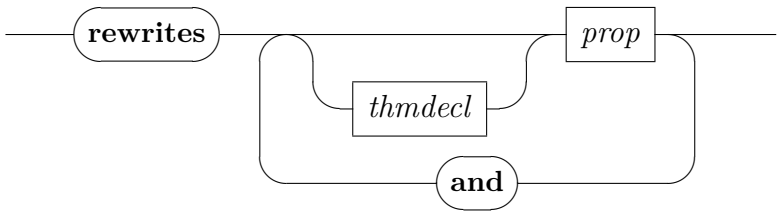
pos_insts



named_insts



rewrites



A locale instance consists of a reference to a locale and either positional or named parameter instantiations optionally followed by rewrites clauses.

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.

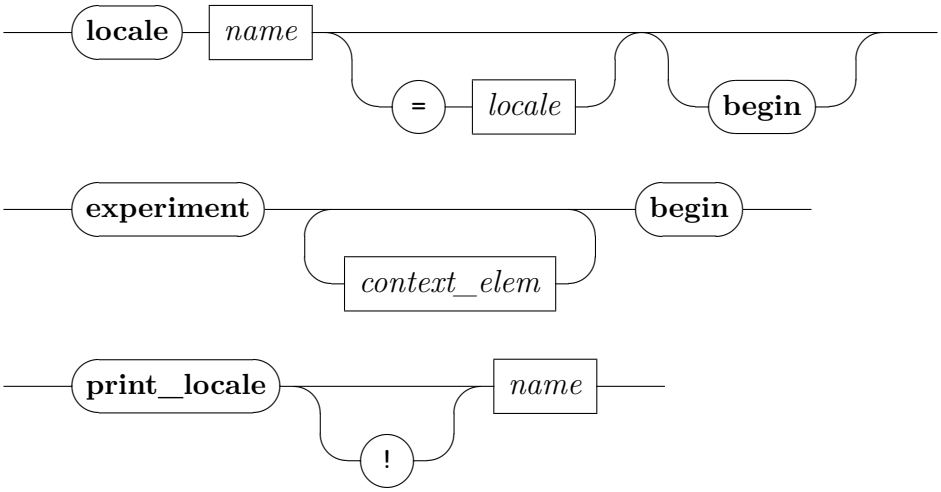
Terms in instantiations are from the context the locale expressions is declared in. Local names may be added to this context with the optional **for** clause. This is useful for shadowing names bound in outer contexts, and for declaring syntax. In addition, syntax declarations from one instance are effective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

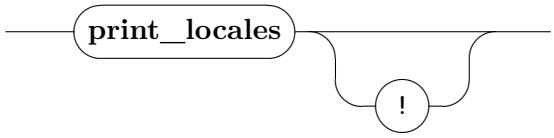
Instances have an optional qualifier which applies to names in declarations. Names include local definitions and theorem names. If present, the qualifier itself is either mandatory (default) or non-mandatory (when followed by "?"). Non-mandatory means that the qualifier may be omitted on input. Qualifiers only affect name spaces; they play no role in determining whether one locale instance subsumes another.

Rewrite clauses amend instances with equations that act as rewrite rules. This is particularly useful for changing concepts introduced through definitions. Rewrite clauses are available only in interpretation commands (see §5.7.3 below) and must be proved the user.

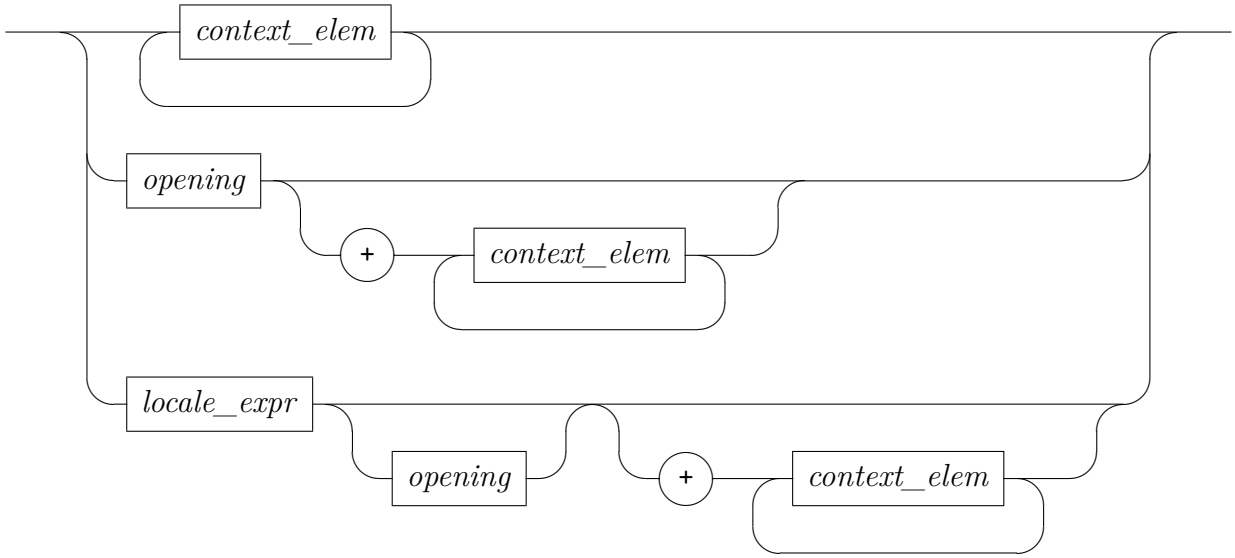
5.7.2 Locale declarations

locale : *theory* → *local_theory*
experiment : *theory* → *local_theory*
print_locale* : *context* →
print_locales* : *context* →
locale_deps* : *context* →

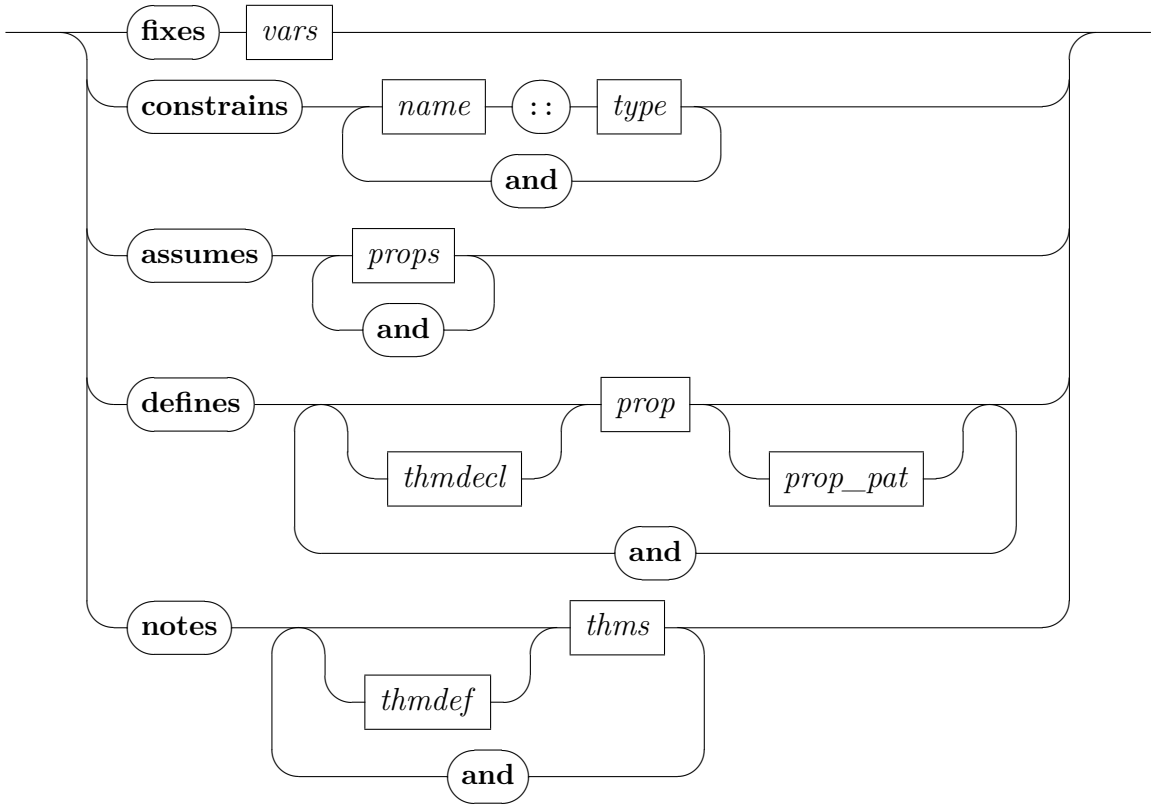



print_locales
!

locale


context_elem
opening
+
context_elem
locale_expr
opening
+
context_elem

context_elem



locale *loc* = *import* **opening** *bundles* + *body* defines a new locale *loc* as a context consisting of a certain view of existing locales (*import*) plus some additional elements (*body*) with declaration *bundles* enriching the context of the command itself. All *import*, *bundles* and *body* are optional; the degenerate form **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 *import* consists of a locale expression; see §5.7.1 above. Its **for** clause defines the parameters of *import*. These are parameters of the defined locale. Locale parameters whose instantiation 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.

Declarations from *bundles*, see §5.3, are effective in the entire command including a subsequent **begin** / **end** block, but they do not contribute to the declarations stored in the locale.

The *body* consists of context elements:

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

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

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

defines a: $x \equiv t$ defines a previously declared parameter. This is similar to **define** within a proof (cf. §6.2.1), but **defines** is restricted to Pure equalities and the defined variable needs to be declared beforehand via **fixes**. The left-hand side of the equation may have additional arguments, e.g. "**defines** $f\ x_1 \ldots x_n \equiv t$", which need to be free in the context.

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.

Both **assumes** and **defines** elements contribute to the locale specification. When defining an operation derived from the parameters, **definition** (§5.4) is usually more appropriate.

Note that "(**is** $p_1 \ldots p_n$)" patterns given in the syntax of **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 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 $\Longrightarrow$ by $\longrightarrow$ in HOL; see also §9.5). Separate introduction rules *loc_axioms.intro* and *loc.intro* are provided as well.

experiment *body* **begin** opens an anonymous locale context with private naming policy. Specifications in its body are inaccessible from outside. This is useful to perform experiments, without polluting the name space.

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

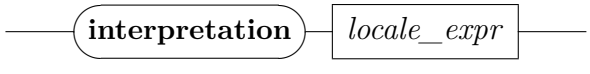
print_locales prints the names of all locales of the current theory; the "!" option indicates extra verbosity.

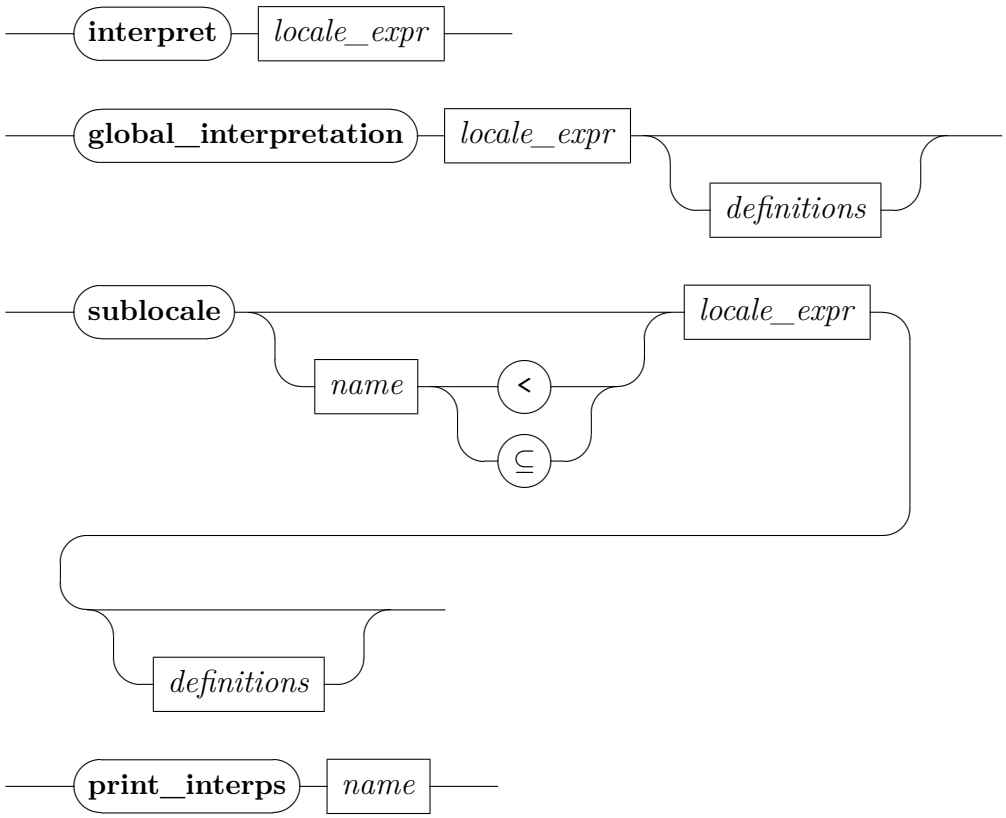
locale_deps visualizes all locales and their relations as a Hasse diagram. This includes locales defined as type classes (§5.8).

5.7.3 Locale interpretation

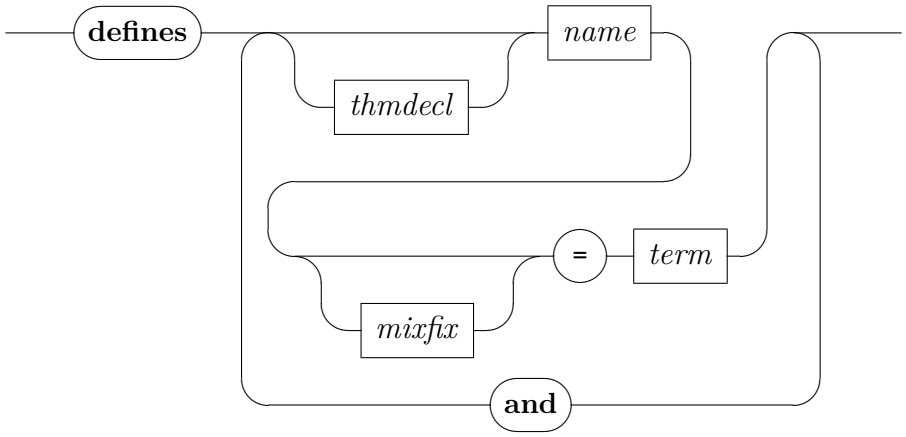
interpretation	:	*local_theory* → *proof*(*prove*)
interpret	:	*proof*(*state*) | *proof*(*chain*) → *proof*(*prove*)
global_interpretation	:	*theory* | *local_theory* → *proof*(*prove*)
sublocale	:	*theory* | *local_theory* → *proof*(*prove*)
print_interps*	:	*context* →
intro_locales	:	*method*
unfold_locales	:	*method*
trace_locales	:	*attribute* default *false*

Locales may be instantiated, and the resulting instantiated declarations added to the current context. This requires proof (of the instantiated specification) and is called *locale interpretation*. Interpretation is possible within arbitrary local theories (**interpretation**), within proof bodies (**interpret**), into global theories (**global_interpretation**) and into locales (**sublocale**).





definitions



The core of each interpretation command is a locale expression *expr*; the command generates proof obligations for the instantiated specifications. Once these are discharged by the user, instantiated declarations (in particular, facts) are added to the context in a post-processing phase, in a manner specific to each command.

Interpretation commands are aware of interpretations that are already active: post-processing is achieved through a variant of roundup that takes interpretations of the current global or local theory into account. In order to simplify the proof obligations according to existing interpretations use methods *intro_locales* or *unfold_locales*.

Rewrites clauses **rewrites** *eqns* occur within expressions. They amend the morphism through which a locale instance is interpreted with rewrite rules, also called rewrite morphisms. This is particularly useful for interpreting concepts introduced through definitions. The equations must be proved the user. To enable syntax of the instantiated locale within the equation, while reading a locale expression, equations of a locale instance are read in a temporary context where the instance is already activated. If activation fails, typically due to duplicate constant declarations, processing falls back to reading the equation first.

Given definitions *defs* produce corresponding definitions in the local theory's underlying target *and* amend the morphism with rewrite rules stemming from the symmetric of those definitions. Hence these need not be proved explicitly the user. Such rewrite definitions are a even more useful device for interpreting concepts introduced through definitions, but they are only supported for interpretation commands operating in a local theory whose implementing target actually supports this. Note that despite the suggestive **and** connective, *defs* are processed sequentially without mutual recursion.

interpretation *expr* interprets *expr* into a local theory such that its lifetime is limited to the current context block (e.g. a locale or unnamed context). At the closing **end** of the block the interpretation and its declarations disappear. Hence facts based on interpretation can be established without creating permanent links to the interpreted locale instances, as would be the case with **sublocale**.

When used on the level of a global theory, there is no end of a current context block, hence **interpretation** behaves identically to **global_interpretation** then.

interpret *expr* interprets *expr* into a proof context: the interpretation and its declarations disappear when closing the current proof block. Note that for **interpret** the *eqns* should be explicitly universally quantified.

global_interpretation *expr* **defines** *defs* interprets *expr* into a global theory.

When adding declarations to locales, interpreted versions of these declarations are added to the global theory for all interpretations in the

global theory as well. That is, interpretations into global theories dynamically participate in any declarations added to locales.

Free variables in the interpreted expression are allowed. They are turned into schematic variables in the generated declarations. In order to use a free variable whose name is already bound in the context — for example, because a constant of that name exists — add it to the **for** clause.

When used in a nested target, resulting declarations are propagated through the whole target stack.

sublocale *name* ⊆ *expr* **defines** *defs* interprets *expr* into 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 locale hierarchy is changed as if *name* imported *expr* (hence the name **sublocale**). When the context of *name* is subsequently entered, traversing the locale hierarchy will involve the locale instances of *expr*, and their declarations will be added to the context. This makes **sublocale** dynamic: extensions of a locale that is instantiated in *expr* may take place after the **sublocale** declaration and still become available in the context. Circular **sublocale** declarations are allowed as long as they do not lead to infinite chains.

If interpretations of *name* exist in the current global theory, the command adds interpretations for *expr* as well, with the same qualifier, although only for fragments of *expr* that are not interpreted in the theory already.

Rewrites clauses in the expression or rewrite definitions *defs* can help break infinite chains induced by circular **sublocale** declarations.

In a named context block the **sublocale** command may also be used, but the locale argument must be omitted. The command then refers to the locale (or class) target of the context block.

print_interps *name* lists all interpretations of locale *name* in the current theory or proof context, including those due to a combination of an **interpretation** or **interpret** and one or several **sublocale** declarations.

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.

While *unfold_locales* is part of the default method for **proof** and often invoked "behind the scenes", *intro_locales* helps understand which proof obligations originated from which locale instances. The latter method is useful while developing proofs but rare in finished developments.

trace_locales, when set to *true*, prints the locale instances activated during roundup. Use this when locale commands yield obscure errors or for understanding local theories created by complex locale hierarchies.

! If a global theory inherits declarations (body elements) for a locale from one parent and an interpretation of that locale from another parent, the interpretation will not be applied to the declarations.

! 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 behaviour of such tools is *not* stable under interpretation morphisms, manual declarations might have to be added to the target context of the interpretation to revert such declarations.

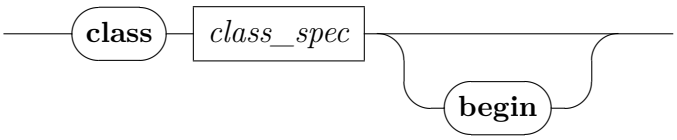
! An interpretation in a local theory or proof context may subsume previous interpretations. This happens if the same specification fragment is interpreted twice and the instantiation of the second interpretation is more general than the interpretation of the first. The locale package does not attempt to remove subsumed interpretations.

! While **interpretation** (**in** *c*) ... is admissible, it is not useful since its result is discarded immediately.

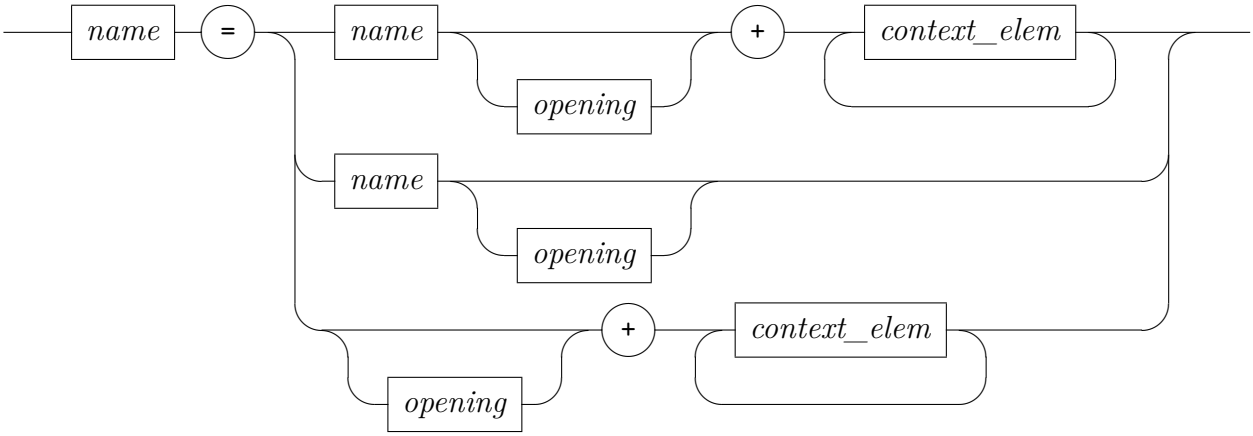
5.8 Classes

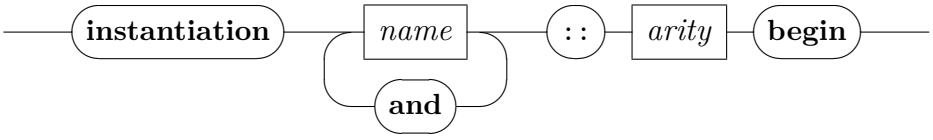
class	:	*theory* → *local_theory*
instantiation	:	*theory* → *local_theory*
instance	:	*local_theory* → *local_theory*
instance	:	*theory* → *proof*(*prove*)
subclass	:	*local_theory* → *local_theory*
print_classes*	:	*context* →
class_deps*	:	*context* →
intro_classes	:	*method*

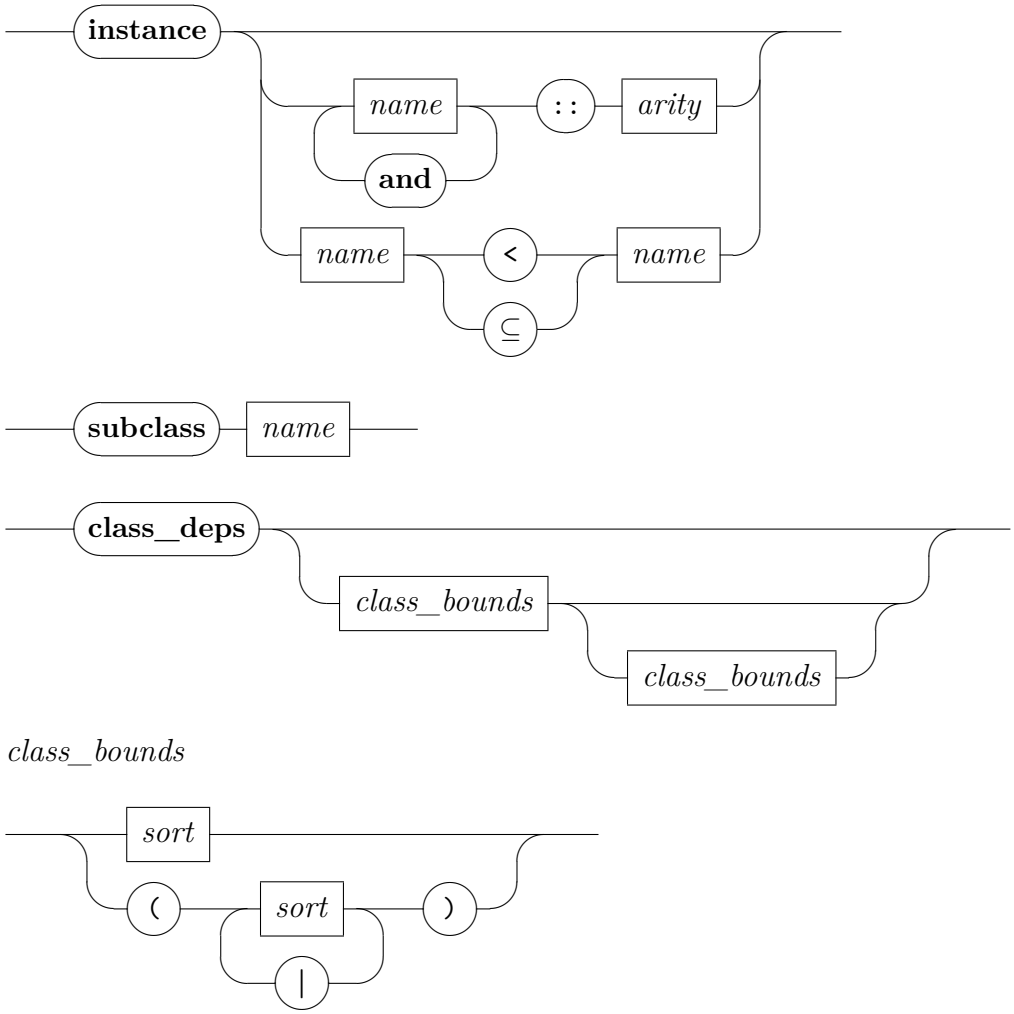
A class is a particular locale with *exactly one* type variable α. Beyond the underlying locale, a corresponding type class is established which is interpreted logically as axiomatic type class [57] whose logical content are the assumptions of the locale. Thus, classes provide the full generality of locales combined with the commodity of type classes (notably type-inference). See [22] for a short tutorial.



class_spec







class c = *superclasses bundles* + *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 α 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.

Optionally given *bundles* take effect in the surface context within the *body* and the potentially following **begin** / **end** block.

instantiation $t :: (s_1, \ldots, s_n)s$ **begin** opens a 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 classes and their subclass relations as a directed acyclic graph. By default, all classes from the current theory context are show. This may be restricted by optional bounds as follows: **class_deps** *upper* or **class_deps** *upper lower*. A class is visualized, iff it is a subclass of some sort from *upper* and a superclass of some sort from *lower*.

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.

5.8.1 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 α 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.

5.8.2 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 $t :: (\overline{s})c$ is completed such that $t :: (\overline{s})c$ and $c \subseteq c'$ implies $t :: (\overline{s})c'$ 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:

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

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 principal types and most general unifiers, e.g. see [40].

5.9 Overloaded constant definitions

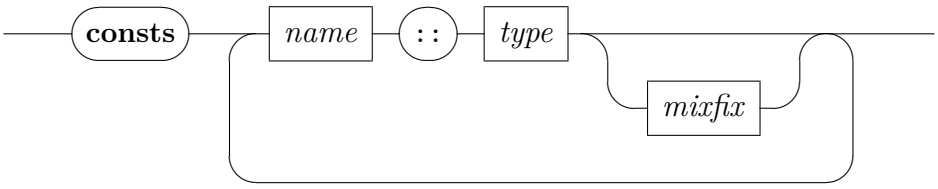
Definitions essentially express abbreviations within the logic. The simplest form of a definition is $c :: \sigma \equiv t$, where c is a new constant and t is a closed term that does not mention c. Moreover, so-called *hidden polymorphism* is excluded: all type variables in t need to occur in its type σ.

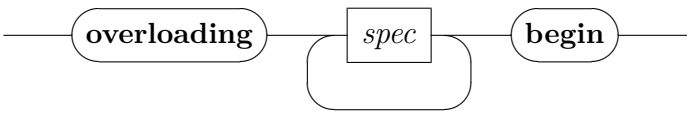
Overloading means that a constant being declared as $c :: \alpha$ *decl* may be defined separately on type instances $c :: (\beta_1, \ldots, \beta_n)\kappa$ *decl* for each type constructor κ. At most occasions overloading will be used in a Haskell-like fashion together with type classes by means of **instantiation** (see §5.8). Sometimes low-level overloading is desirable; this is supported by **consts** and **overloading** explained below.

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, and general $d :: \alpha \times \beta$ will be disallowed. Full details are given by Kunčar [27].

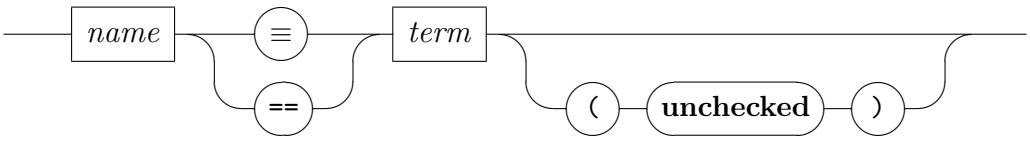
The **consts** command and the **overloading** target provide a convenient interface for end-users. Regular specification elements such as **definition**, **inductive**, **function** may be used in the body. It is also possible to use **consts** $c :: \sigma$ with later **overloading** $c \equiv c :: \sigma$ to keep the declaration and definition of a constant separate.

consts : *theory* → *theory*
overloading : *theory* → *local_theory*





spec



consts $c :: \sigma$ declares constant c to have any instance of type scheme σ. The optional mixfix annotations may attach concrete syntax to the constants declared.

overloading $x_1 \equiv c_1 :: \tau_1 \ \ldots \ x_n \equiv c_n :: \tau_n$ **begin** ... **end** defines a theory target (cf. §5.2) which allows to specify already declared constants via definitions in the body. 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.

Option (**unchecked**) disables global dependency checks for the corresponding definition, which is occasionally useful for exotic overloading; this is a form of axiomatic specification. It is at the discretion of the user to avoid malformed theory specifications!

Example

consts $Length :: 'a \Rightarrow nat$

overloading
$Length_0 \equiv Length :: unit \Rightarrow nat$
$Length_1 \equiv Length :: 'a \times unit \Rightarrow nat$
$Length_2 \equiv Length :: 'a \times 'b \times unit \Rightarrow nat$
$Length_3 \equiv Length :: 'a \times 'b \times 'c \times unit \Rightarrow nat$
begin

fun $Length_0 :: unit \Rightarrow nat$ **where** $Length_0\ () = 0$
fun $Length_1 :: 'a \times unit \Rightarrow nat$ **where** $Length_1\ (a, ()) = 1$
fun $Length_2 :: 'a \times 'b \times unit \Rightarrow nat$ **where** $Length_2\ (a, b, ()) = 2$
fun $Length_3 :: 'a \times 'b \times 'c \times unit \Rightarrow nat$ **where** $Length_3\ (a, b, c, ()) = 3$

end

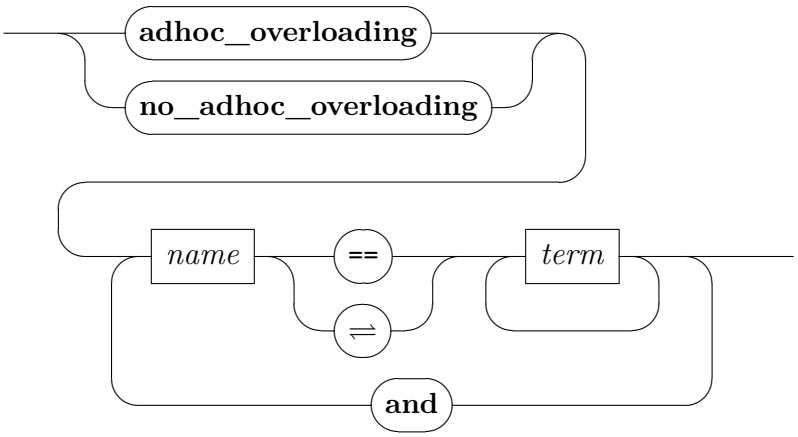
lemma $Length\ (a, b, c, ()) = 3$ **by** *simp*

lemma *Length* ((*a*, *b*), (*c*, *d*), ()) = 2 **by** *simp*
lemma *Length* ((*a*, *b*, *c*, *d*, *e*), ()) = 1 **by** *simp*

5.10 Overloaded constant abbreviations: adhoc overloading

adhoc_overloading	:	*local_theory* → *local_theory*	
no_adhoc_overloading	:	*local_theory* → *local_theory*	
show_variants	:	*attribute*	default *false*

Adhoc overloading allows to overload a constant depending on its type. Typically this involves the introduction of an uninterpreted constant (used for input and output) and the addition of some variants (used internally). For examples see `~~/src/HOL/Examples/Adhoc_Overloading.thy` and `~~/src/HOL/Library/Monad_Syntax.thy`.



adhoc_overloading $c \rightleftharpoons v_1 \ldots v_n$ associates variants with an existing constant.

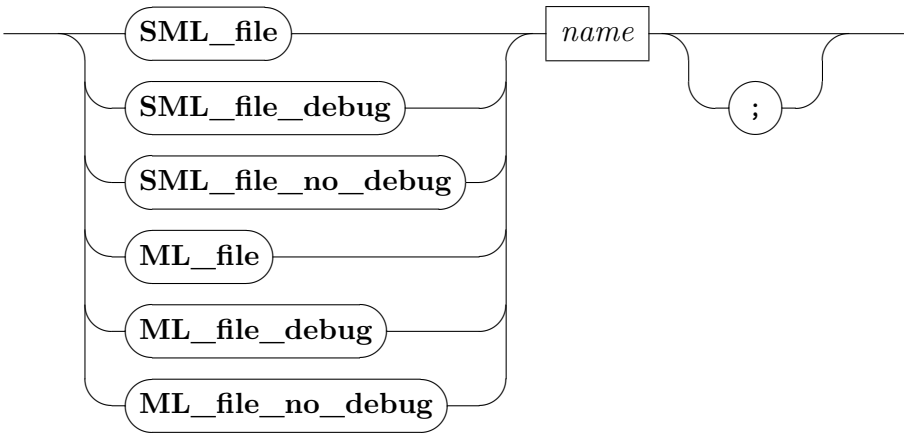
no_adhoc_overloading is similar to **adhoc_overloading**, but removes the specified variants from the present context.

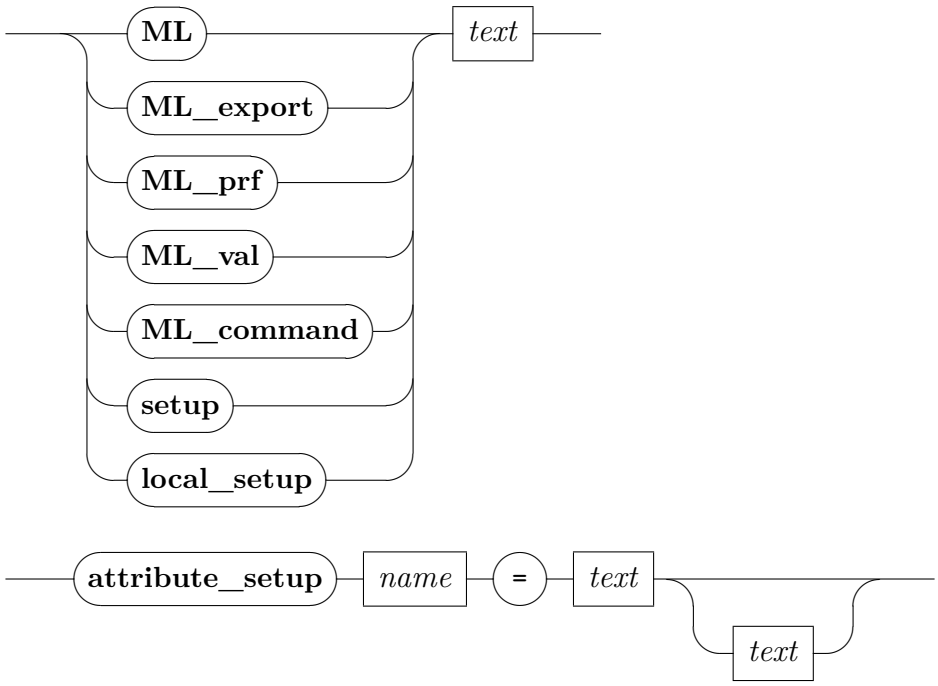
show_variants controls printing of variants of overloaded constants. If enabled, the internally used variants are printed instead of their respective overloaded constants. This is occasionally useful to check whether the system agrees with a user's expectations about derived variants.

5.11 Incorporating ML code

SML_file : *local_theory* → *local_theory*
SML_file_debug : *local_theory* → *local_theory*
SML_file_no_debug : *local_theory* → *local_theory*
ML_file : *local_theory* → *local_theory*
ML_file_debug : *local_theory* → *local_theory*
ML_file_no_debug : *local_theory* → *local_theory*
ML : *local_theory* → *local_theory*
ML_export : *local_theory* → *local_theory*
ML_prf : *proof* → *proof*
ML_val : *any* →
ML_command : *any* →
setup : *theory* → *theory*
local_setup : *local_theory* → *local_theory*
attribute_setup : *local_theory* → *local_theory*

ML_print_depth : *attribute* default 10
ML_source_trace : *attribute* default *false*
ML_debugger : *attribute* default *false*
ML_exception_trace : *attribute* default *false*
ML_exception_debugger : *attribute* default *false*
ML_environment : *attribute* default *Isabelle*





SML_file *name* reads and evaluates the given Standard ML file. Top-level SML bindings are stored within the (global or local) theory context; the initial environment is restricted to the Standard ML implementation of Poly/ML, without the many add-ons of Isabelle/ML. Multiple **SML_file** commands may be used to build larger Standard ML projects, independently of the regular Isabelle/ML environment.

ML_file *name* reads and evaluates the given ML file. The current theory context is passed down to the ML toplevel and may be modified, using `Context.>>` or derived ML commands. Top-level ML bindings are stored within the (global or local) theory context.

SML_file_debug, **SML_file_no_debug**, **ML_file_debug**, and **ML_file_no_debug** change the *ML_debugger* option locally while the given file is compiled.

ML is similar to **ML_file**, but evaluates directly the given *text*. Top-level ML bindings are stored within the (global or local) theory context.

ML_export is similar to **ML**, but the resulting toplevel bindings are exported to the global bootstrap environment of the ML process — it has a lasting effect that cannot be retracted. This allows ML evaluation without a formal theory context, e.g. for command-line tools via `isabelle ML_process` [54].

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 context. 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 ths
        (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))›
```

ML_print_depth controls the printing depth of the ML toplevel pretty printer. Typically the limit should be less than 10. Bigger values such as 100–1000 are occasionally useful for debugging.

ML_source_trace indicates whether the source text that is given to the ML compiler should be output: it shows the raw Standard ML after expansion of Isabelle/ML antiquotations.

ML_debugger controls compilation of sources with or without debugging information. The global system option `ML_debugger` does the same when building a session image. It is also possible use commands like **ML_file_debug** etc. The ML debugger is explained further in [56].

ML_exception_trace indicates whether the ML run-time system should print a detailed stack trace on exceptions. The result is dependent on various ML compiler optimizations. The boundary for the exception trace is the current Isar command transactions: it is occasionally better to insert the combinator `Runtime.exn_trace` into ML code for debugging [55], closer to the point where it actually happens.

ML_exception_debugger controls detailed exception trace via the Poly/ML debugger, at the cost of extra compile-time and run-time overhead. Relevant ML modules need to be compiled beforehand with debugging enabled, see *ML_debugger* above.

ML_environment determines the named ML environment for toplevel declarations, e.g. in command **ML** or **ML_file**. The following ML environments are predefined in Isabelle/Pure:

- *Isabelle* for Isabelle/ML. It contains all modules of Isabelle/Pure and further add-ons, e.g. material from Isabelle/HOL.
- *SML* for official Standard ML. It contains only the initial basis according to http://sml-family.org/Basis/overview.html.

The Isabelle/ML function `ML_Env.setup` defines a new ML environment. This is useful to incorporate big SML projects in an isolated name space, possibly with variations on ML syntax; the existing setup of `ML_Env.SML_operations` follows the official standard.

It is also possible to move toplevel bindings between ML environments, using a notation with "`>`" as separator. For example:

declare [[*ML_environment* = *Isabelle*>*SML*]]
ML ‹*val println = writeln*›

declare [[*ML_environment* = *SML*]]
ML ‹*println "test"*›

declare [[*ML_environment* = *Isabelle*]]
ML ‹*ML ‹println› (*bad*) handle ERROR msg => warning msg*›

5.12 Generated files and exported files

Write access to the physical file-system is incompatible with the stateless model of processing Isabelle documents. To avoid bad effects, the following concepts for abstract file-management are provided by Isabelle:

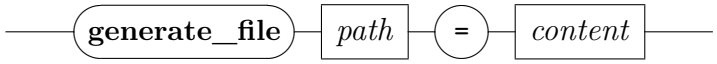
Generated files are stored within the theory context in Isabelle/ML. This allows to operate on the content in Isabelle/ML, e.g. via the command **compile_generated_files**.

Exported files are stored within the session database in Isabelle/Scala. This allows to deliver artefacts to external tools, see also [54] for session `ROOT` declaration **export_files**, and `isabelle build` option `-e`.

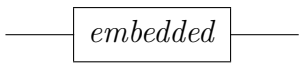
A notable example is the command **export_code** (chapter 13): it uses both concepts simultaneously.

File names are hierarchically structured, using a slash as separator. The (long) theory name is used as a prefix: the resulting name needs to be globally unique.

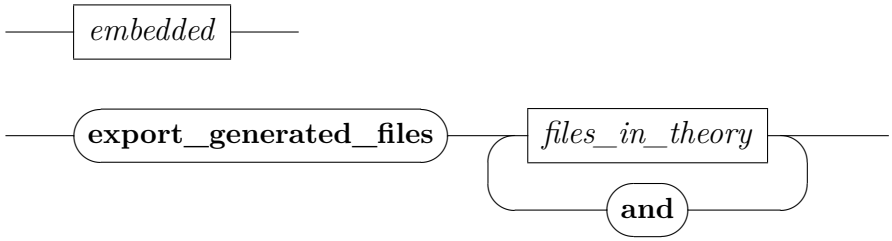
generate_file	:	*local_theory* → *local_theory*
export_generated_files	:	*context* →
compile_generated_files	:	*context* →
external_file	:	*any* → *any*



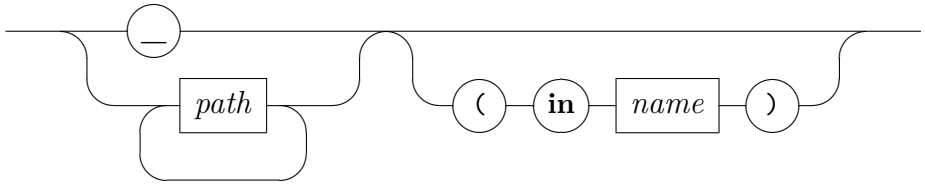
path

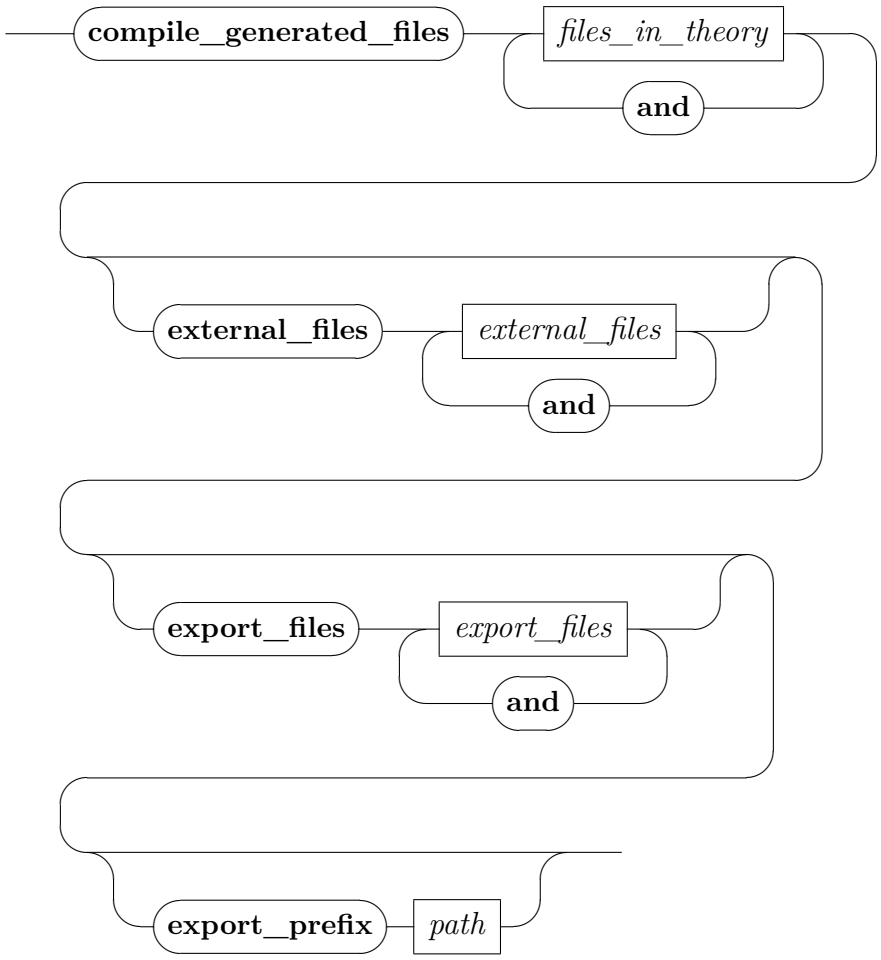


content

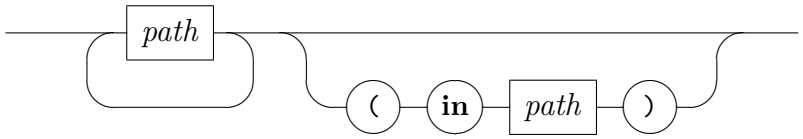


files_in_theory

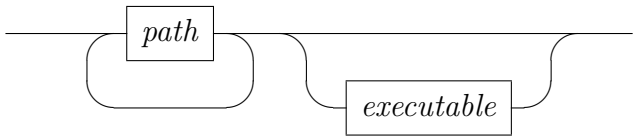




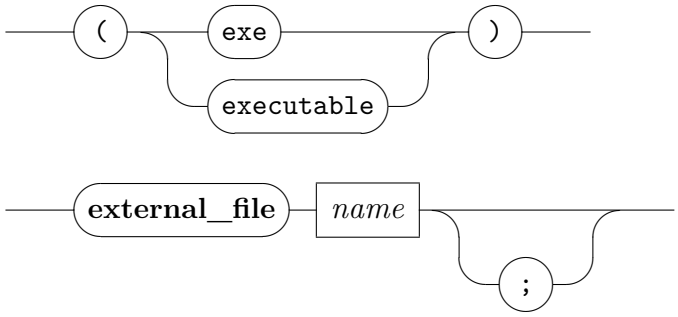
external_files



export_files



executable



generate_file *path* = *content* augments the table of generated files within the current theory by a new entry: duplicates are not allowed. The name extension determines a pre-existent file-type; the *content* is a string that is preprocessed according to rules of this file-type.

For example, Isabelle/Pure supports `.hs` as file-type for Haskell: embedded cartouches are evaluated as Isabelle/ML expressions of type `string`, the result is inlined in Haskell string syntax.

export_generated_files *paths* (**in** *thy*) retrieves named generated files from the given theory (that needs be reachable via imports of the current one). By default, the current theory node is used. Using "`_`" (underscore) instead of explicit path names refers to *all* files of a theory node.

The overall list of files is prefixed with the respective (long) theory name and exported to the session database. In Isabelle/jEdit the result can be browsed via the virtual file-system with prefix "`isabelle-export:`" (using the regular file-browser).

scala_build_generated_files *paths* (**in** *thy*) retrieves named generated files as for **export_generated_files** and writes them into a temporary directory, which is taken as starting point for build process of Isabelle/Scala/Java modules (see [54]). The corresponding `build.props` file is expected directly in the toplevel directory, instead of `etc/build.props` for Isabelle system components. These properties need to specify sources, resources, services etc. as usual. The resulting JAR module becomes an export artefact of the session database, with a name of the form "*theory*`:classpath/`*module*`.jar`".

compile_generated_files *paths* (**in** *thy*) **where** *compile_body* retrieves named generated files as for **export_generated_files** and writes

them into a temporary directory, such that the *compile_body* may operate on them as an ML function of type `Path.T -> unit`. This may create further files, e.g. executables produced by a compiler that is invoked as external process (e.g. via `Isabelle_System.bash`), or any other files.

The option "**external_files** *paths* (**in** *base_dir*)" copies files from the physical file-system into the temporary directory, *before* invoking *compile_body*. The *base_dir* prefix is removed from each of the *paths*, but the remaining sub-directory structure is reconstructed in the target directory.

The option "**export_files** *paths*" exports the specified files from the temporary directory to the session database, *after* invoking *compile_body*. Entries may be decorated with "(*exe*)" to say that it is a platform-specific executable program: the executable file-attribute will be set, and on Windows the `.exe` file-extension will be included; "(*executable*)" only refers to the file-attribute, without special treatment of the `.exe` extension.

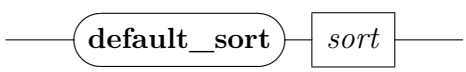
The option "**export_prefix** *path*" specifies an extra path prefix for all exports of **export_files** above.

external_file *name* declares the formal dependency on the given file name, such that the Isabelle build process knows about it (see also [54]). This is required for any files mentioned in **compile_generated_files** / **external_files** above, in order to document source dependencies properly. It is also possible to use **external_file** alone, e.g. when other Isabelle/ML tools use `File.read`, without specific management of content by the Prover IDE.

5.13 Primitive specification elements

5.13.1 Sorts

default_sort : *local_theory* → *local_theory*



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

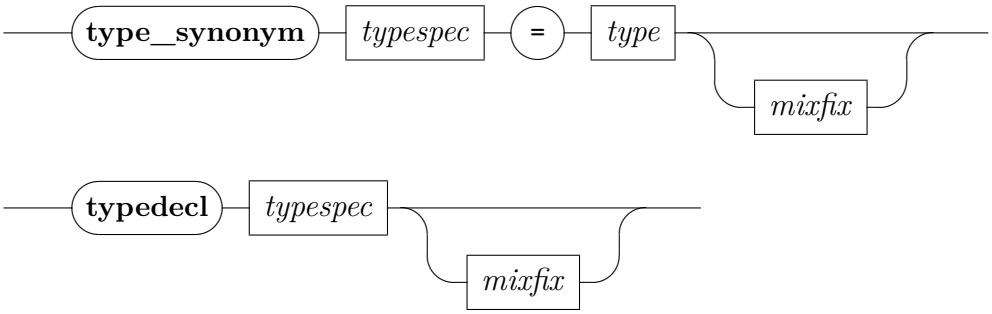
current context). Type variables generated by type inference are not affected.

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.

5.13.2 Types

type_synonym : *local_theory* → *local_theory*
typedecl : *local_theory* → *local_theory*



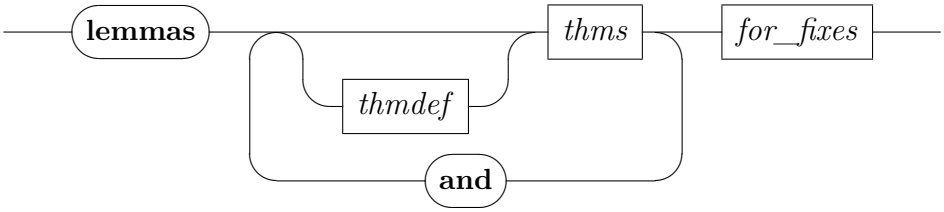
type_synonym $(\alpha_1, \ldots, \alpha_n)\ t = \tau$ introduces a *type synonym* $(\alpha_1, \ldots, \alpha_n)\ t$ for the existing type τ. Unlike the semantic type definitions in Isabelle/HOL, type synonyms are merely syntactic abbreviations without any logical significance. Internally, type synonyms are fully expanded.

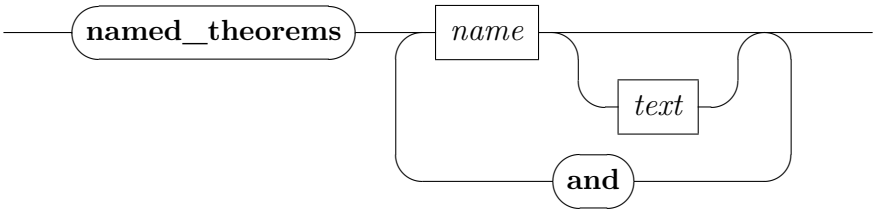
typedecl $(\alpha_1, \ldots, \alpha_n)\ t$ declares a new type constructor t. If the object-logic defines a base sort s, then the constructor is declared to operate on that, via the axiomatic type-class instance $t :: (s, \ldots, s)s$.

! If you introduce a new type axiomatically, i.e. via **typedecl** and **axiomatization** (§5.5), the minimum requirement is that it has a non-empty model, to avoid immediate collapse of the logical environment. Moreover, one needs to demonstrate that the interpretation of such free-form axiomatizations can coexist with other axiomatization schemes for types, notably **typedef** in Isabelle/HOL (§11.6), or any other extension that people might have introduced elsewhere.

5.14 Naming existing theorems

lemmas : *local_theory* → *local_theory*
named_theorems : *local_theory* → *local_theory*





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.

named_theorems *name description* declares a dynamic fact within the context. The same *name* is used to define an attribute with the usual *add*/*del* syntax (e.g. see §9.3.2) to maintain the content incrementally, in canonical declaration order of the text structure.

5.15 Oracles

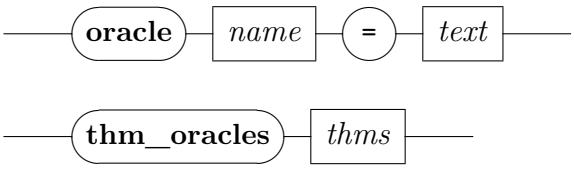
oracle : *theory* → *theory* (*axiomatic*!)
thm_oracles* : *context* →

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. This also includes implicit type-class reasoning via the order-sorted algebra of class relations and type arities (see also **instantiation** and **instance**).



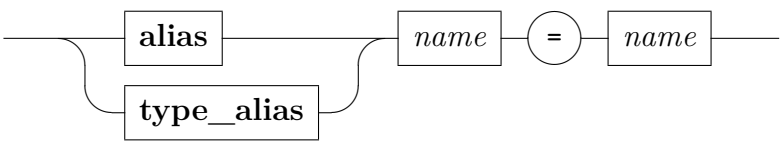
oracle *name* = *text* turns the given ML expression *text* of type `'a -> cterm` into an ML function of type `'a -> thm`, which is bound to the global identifier `name`. This acts like an infinitary specification of axioms! Invoking the oracle only works within the scope of the resulting theory.

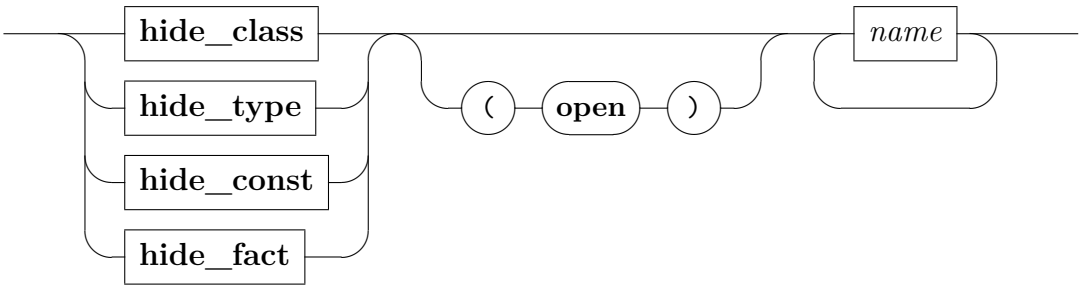
See `~~/src/HOL/Examples/Iff_Oracle.thy` for a worked example of defining a new primitive rule as oracle, and turning it into a proof method.

thm_oracles *thms* displays all oracles used in the internal derivation of the given theorems; this covers the full graph of transitive dependencies.

5.16 Name spaces

alias	:	*local_theory* → *local_theory*
type_alias	:	*local_theory* → *local_theory*
hide_class	:	*theory* → *theory*
hide_type	:	*theory* → *theory*
hide_const	:	*theory* → *theory*
hide_fact	:	*theory* → *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 behaviour of name spaces by hand, yet the following commands provide some way to do so.

alias and **type_alias** introduce aliases for constants and type constructors, respectively. This allows adhoc changes to name-space accesses.

type_alias $b = c$ introduces an alias for an existing type constructor.

hide_class *names* fully removes class declarations from a given name space; with the (*open*) option, only the unqualified 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.

Chapter 6

Proofs

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.

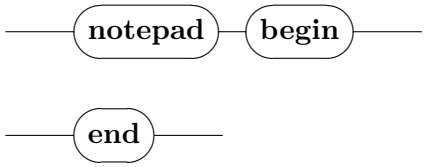
The proof mode indicator may be understood as an instruction to the writer, telling what kind of operation may be performed next. The corresponding typings of proof commands restricts the shape of well-formed proof texts to particular command sequences. So dynamic arrangements of commands eventually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

notepad : *local_theory* → *proof*(*state*)



notepad begin opens a proof state without any goal statement. This allows to experiment with Isar, without producing any persistent result. The notepad is closed by **end**.

6.1.2 Blocks

$$\begin{array}{rcl} \textbf{next} & : & \mathit{proof}(\mathit{state}) \to \mathit{proof}(\mathit{state}) \\ \textbf{\{} & : & \mathit{proof}(\mathit{state}) \to \mathit{proof}(\mathit{state}) \\ \textbf{\}} & : & \mathit{proof}(\mathit{state}) \to \mathit{proof}(\mathit{state}) \end{array}$$

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

oops : *proof* → *local_theory* | *theory*

The **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 **sorry** (see §6.4.2): **oops** does not observe the proof structure at all, but goes back right to the theory level. Furthermore, **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 **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 LaTeX macros can be easily adapted to print something like "..." instead of the keyword "**oops**".

6.2 Statements

6.2.1 Context elements

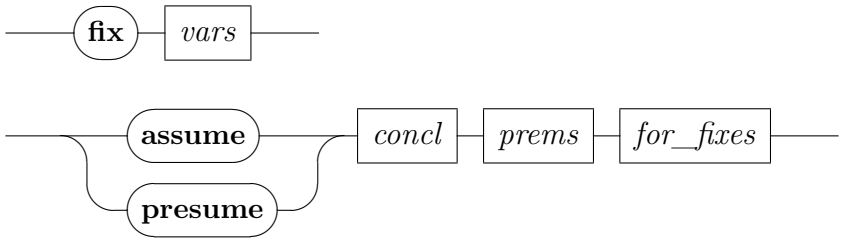
fix : $\mathit{proof}(\mathit{state}) \to \mathit{proof}(\mathit{state})$
assume : $\mathit{proof}(\mathit{state}) \to \mathit{proof}(\mathit{state})$
presume : $\mathit{proof}(\mathit{state}) \to \mathit{proof}(\mathit{state})$
define : $\mathit{proof}(\mathit{state}) \to \mathit{proof}(\mathit{state})$

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: $\vdash \bigwedge x.\ \varphi[x]$ (this is expressed in normal form using Isabelle's meta-variables).

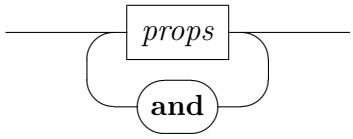
Similarly, introducing some assumption χ has two effects. On the one hand, a local theorem is created that may be used as a fact in subsequent proof steps. On the other hand, any result $\chi \vdash \varphi$ exported from the context becomes conditional wrt. the assumption: $\vdash \chi \Longrightarrow \varphi$. 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 **assume** insists on solving the subgoal

by unification with some premise of the goal, **presume** leaves the subgoal unchanged in order to be proved later by the user.

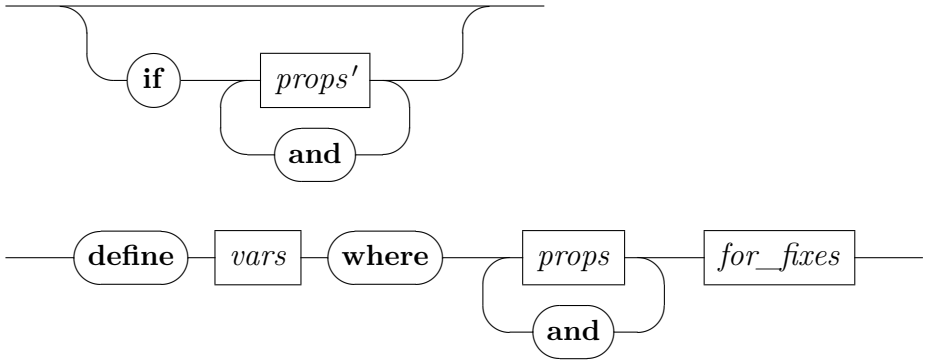
Local definitions, introduced by "**define** x **where** $x = t$", are achieved by combining "**fix** x" with another version of assumption that causes any hypothetical equation $x \equiv t$ to be eliminated by the reflexivity rule. Thus, exporting some result $x \equiv t \vdash \varphi[x]$ yields $\vdash \varphi[t]$.



concl



prems



fix x introduces a local variable x that is *arbitrary, but fixed.*

assume a: φ and **presume** a: φ introduce a local fact $\varphi \vdash \varphi$ 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 φ 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.

A structured assumption like **assume** $B\ x$ **if** $A\ x$ **for** x is equivalent to **assume** $\bigwedge x.\ A\ x \Longrightarrow B\ x$, but vacuous quantification is avoided: a for-context only effects propositions according to actual use of variables.

define x **where** $x = t$ introduces a local (non-polymorphic) definition. In results that are exported from the context, x is replaced by t.

Internally, equational assumptions are added to the context in Pure form, using $x \equiv t$ instead of $x = t$ or $x \longleftrightarrow t$ from the object-logic. When exporting results from the context, x is generalized and the assumption discharged by reflexivity, causing the replacement by t.

The default name for the definitional fact is *x_def*. Several simultaneous definitions may be given as well, with a collective default name.

It is also possible to abstract over local parameters as follows: **define** $f :: 'a \Rightarrow 'b$ **where** $f\ x = t$ **for** $x :: 'a$.

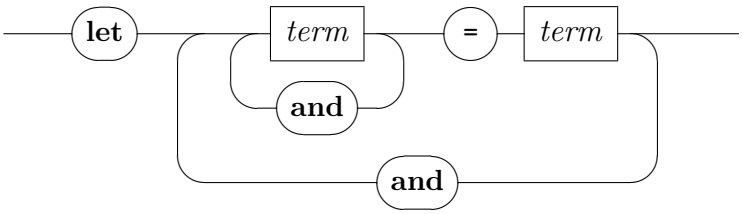
6.2.2 Term abbreviations

let : *proof*(*state*) → *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 ... 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 **define** (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 **define** does not support polymorphism.



The syntax of **is** patterns follows *term_pat* or *prop_pat* (see §3.3.8).

let $p_1 = t_1$ **and** ... $p_n = t_n$ binds any text variables in patterns p_1, ..., p_n by simultaneous higher-order matching against terms t_1, ..., t_n.

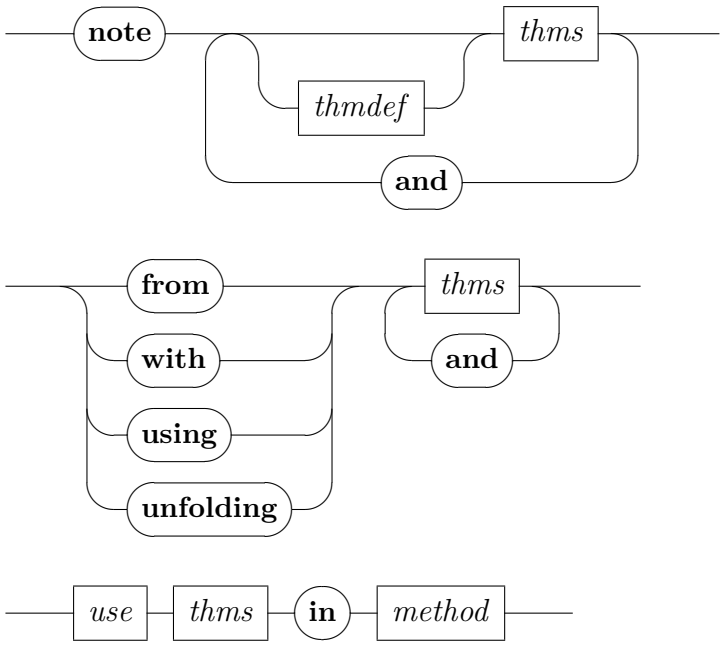
(**is** p_1 ... p_n) resembles **let**, but matches p_1, ..., p_n against the preceding statement. Also note that **is** is not a separate command, but part of others (such as **assume**, **have** etc.).

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 §6.3).

6.2.3 Facts and forward chaining

note	:	$proof(state) \to proof(state)$
then	:	$proof(state) \to proof(chain)$
from	:	$proof(state) \to proof(chain)$
with	:	$proof(state) \to proof(chain)$
using	:	$proof(prove) \to proof(prove)$
unfolding	:	$proof(prove) \to proof(prove)$
use	:	*method*
method_facts	:	*fact*

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.



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 §6.4.2). For example, method *rule* (see §6.4.3) 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 to be used 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. See also the proof method *unfold*.

(*use* b_1 ... b_n **in** *method*) uses the facts in the given method expression. The facts provided by the proof state (via **using** etc.) are ignored, but it is possible to refer to *method_facts* explicitly.

method_facts is a dynamic fact that refers to the currently used facts of the goal state.

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 $PROP\ \psi \Longrightarrow PROP\ \psi$, 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.

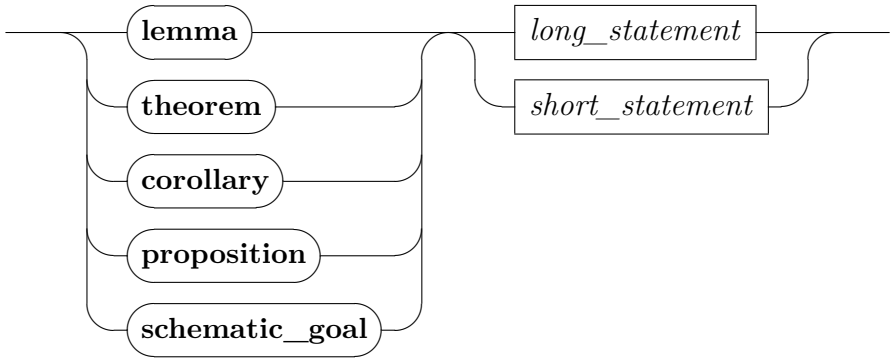
6.2.4 Goals

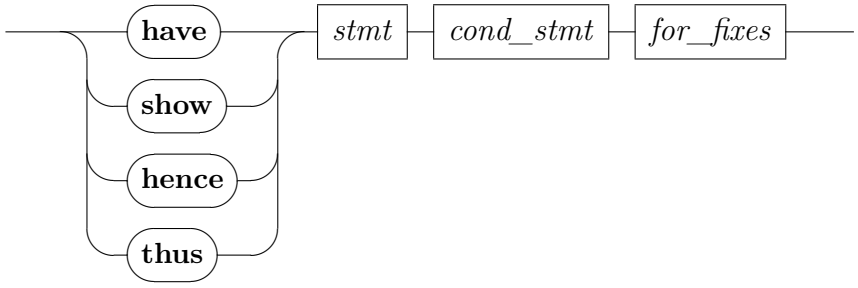
lemma	:	*local_theory* → *proof*(*prove*)
theorem	:	*local_theory* → *proof*(*prove*)
corollary	:	*local_theory* → *proof*(*prove*)
proposition	:	*local_theory* → *proof*(*prove*)
schematic_goal	:	*local_theory* → *proof*(*prove*)
have	:	*proof*(*state*) | *proof*(*chain*) → *proof*(*prove*)
show	:	*proof*(*state*) | *proof*(*chain*) → *proof*(*prove*)
hence	:	*proof*(*state*) → *proof*(*prove*)
thus	:	*proof*(*state*) → *proof*(*prove*)
print_statement*	:	*context* →

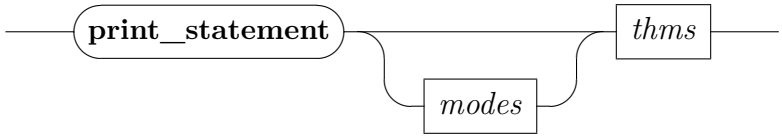
From a theory context, proof mode is entered by an initial goal command such as **lemma**. Within a proof context, new claims may be introduced locally; there are variants to interact with the overall proof structure specifically, such as **have** or **show**.

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** (§6.4.2) or **apply** (§7.1). The *induct* method covered in §6.5 acts on multiple claims simultaneously.

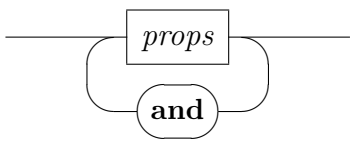
Claims at the theory level may be either in short or long form. A short goal merely consists of several simultaneous propositions (often just one). A long goal includes an explicit context specification for the subsequent conclusion, involving local parameters and assumptions. Here the role of each part of the statement is explicitly marked by separate keywords (see also §5.7); 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 contexts — essentially a big disjunction of eliminated parameters and assumptions (see §6.6).



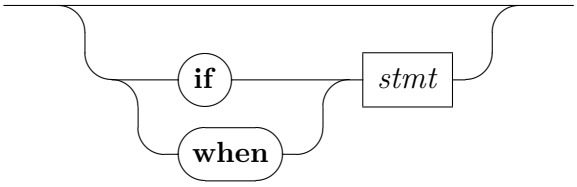




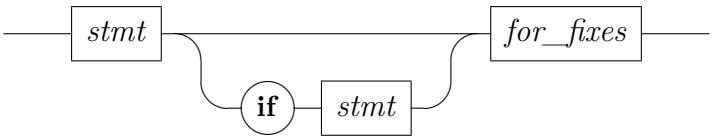
stmt



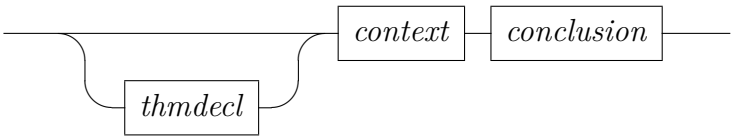
cond_stmt



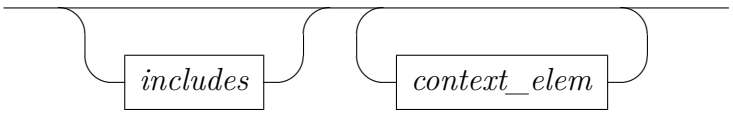
short_statement



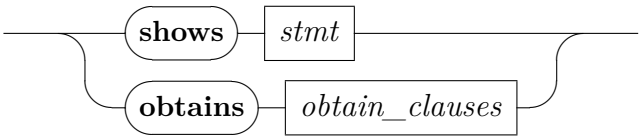
long_statement



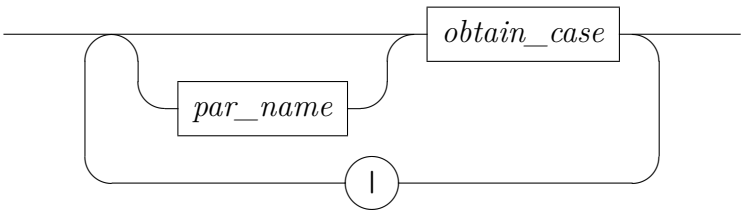
context



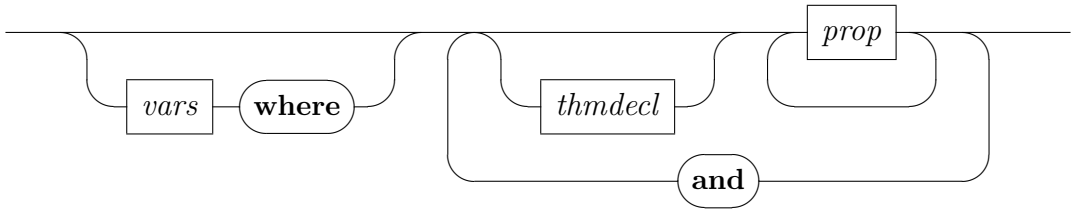
conclusion



obtain_clauses



obtain_case



lemma *a*: φ enters proof mode with φ as main goal, eventually resulting in some fact $\vdash \varphi$ to be put back into the target context.

A *long_statement* may build up an initial proof context for the subsequent claim, potentially including local definitions and syntax; see also *includes* in §5.3 and *context_elem* in §5.7.

A *short_statement* consists of propositions as conclusion, with an option context of premises and parameters, via `if`/`for` in postfix notation, corresponding to `assumes`/`fixes` in the long prefix notation.

Local premises (if present) are called "*assms*" for *long_statement*, and "*that*" for *short_statement*.

theorem, **corollary**, and **proposition** are the same as **lemma**. The different command names merely serve as a formal comment in the theory source.

schematic_goal is similar to **theorem**, but allows 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*: φ 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*: φ is like **have** *a*: φ 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:

```
Local statement fails to refine any pending goal
```

hence expands to "**then have**" and **thus** expands to "**then show**". These conflations are left-over from early history of Isar. The expanded syntax is more orthogonal and improves readability and maintainability of proofs.

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 *?thesis*) to be bound automatically, see also §6.2.2.

Structured goal statements involving **if** or **when** define the special fact *that* to refer to these assumptions in the proof body. The user may provide separate names according to the syntax of the statement.

6.3 Calculational reasoning

also	:	$proof(state) \to proof(state)$
finally	:	$proof(state) \to proof(chain)$
moreover	:	$proof(state) \to proof(state)$
ultimately	:	$proof(state) \to proof(chain)$
print_trans_rules*	:	$context \to$
trans	:	*attribute*
sym	:	*attribute*
symmetric	:	*attribute*

Calculational proof is forward reasoning with implicit application of transitivity rules (such those of $=$, $\leq$, $<$). Isabelle/Isar maintains an auxiliary fact register *calculation* for accumulating results obtained by transitivity composed with the current result. Command **also** updates *calculation* involving *this*, while **finally** exhibits the final *calculation* by forward chaining towards the next goal statement. Both commands require valid current facts, i.e. may

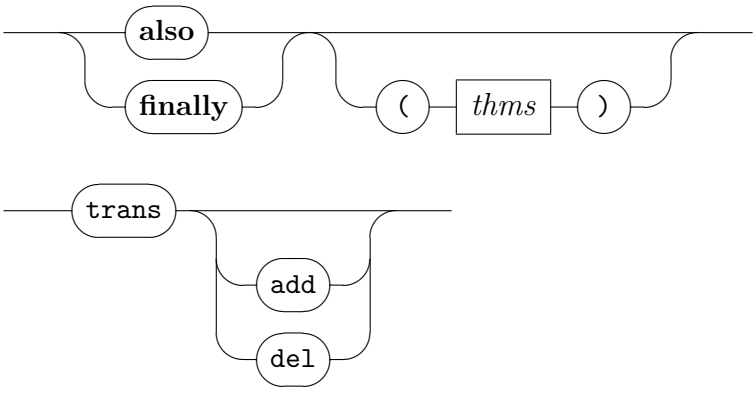
occur only after commands that produce theorems such as **assume**, **note**, or some finished proof of **have**, **show** etc. The **moreover** and **ultimately** commands are similar to **also** and **finally**, but only collect further results in *calculation* without applying any rules yet.

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 *begin-calculation* command required.

The Isar calculation proof commands may be defined as follows:[1]

$$\begin{aligned}
\textbf{also}_0 &\equiv \textbf{note}\ \mathit{calculation} = \mathit{this} \\
\textbf{also}_{n+1} &\equiv \textbf{note}\ \mathit{calculation} = \mathit{trans}\ [\mathit{OF}\ \mathit{calculation}\ \mathit{this}] \\
\textbf{finally} &\equiv \textbf{also from}\ \mathit{calculation} \\
\textbf{moreover} &\equiv \textbf{note}\ \mathit{calculation} = \mathit{calculation}\ \mathit{this} \\
\textbf{ultimately} &\equiv \textbf{moreover from}\ \mathit{calculation}
\end{aligned}$$



also (a_1 ... a_n) maintains the auxiliary *calculation* register as follows. The first occurrence of **also** in some calculational thread initializes *calculation* by *this*. Any subsequent **also** on the same level of block-structure updates *calculation* by some transitivity rule applied to

[1] We suppress internal bookkeeping such as proper handling of block-structure.

calculation and *this* (in that order). Transitivity rules are picked from the current context, unless alternative rules are given as explicit arguments.

finally (a_1 ... a_n) maintains *calculation* in the same way as **also** and then concludes the current calculational thread. The final result is exhibited as fact for forward chaining towards the next goal. Basically, **finally** abbreviates **also from** *calculation*. Typical idioms for concluding calculational proofs are "**finally show** *?thesis* **.**" and "**finally have** φ **.**".

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 patters) 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** [*symmetric*]: $x = y$" produces a swapped fact derived from that assumption.

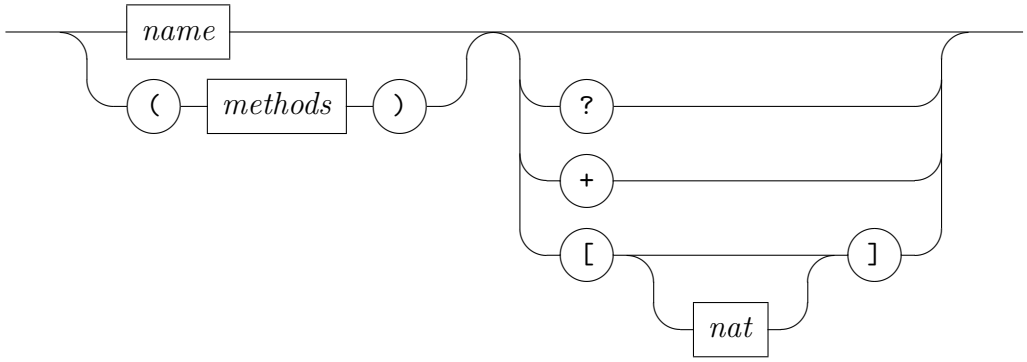
In structured proof texts it is often more appropriate to use an explicit single-step elimination proof, such as "**assume** $x = y$ **then have** $y = x$ **..**".

6.4 Refinement steps

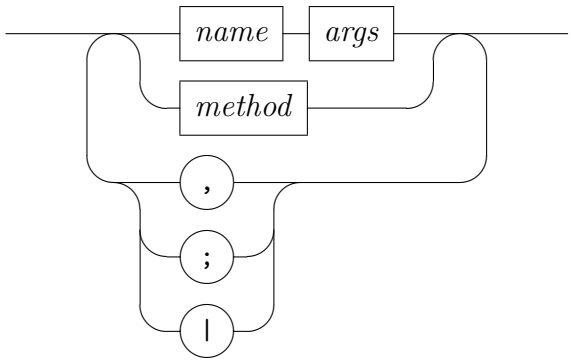
6.4.1 Proof method expressions

Proof methods are either basic ones, or expressions composed of methods via "`,`" (sequential composition), "`;`" (structural composition), "`|`" (alternative choices), "`?`" (try), "`+`" (repeat at least once), "`[`*n*`]`" (restriction to first n subgoals). In practice, proof methods are usually just a comma separated list of *name args* specifications. Note that parentheses may be dropped for single method specifications (with no arguments). The syntactic precedence of method combinators is `| ; , [] + ?` (from low to high).

method



methods



Regular Isar proof methods do *not* admit direct goal addressing, but refer to the first subgoal or to all subgoals uniformly. Nonetheless, the subsequent mechanisms allow to imitate the effect of subgoal addressing that is known from ML tactics.

Goal *restriction* means the proof state is wrapped-up in a way that certain subgoals are exposed, and other subgoals are "parked" elsewhere. Thus a proof method has no other chance than to operate on the subgoals that are presently exposed.

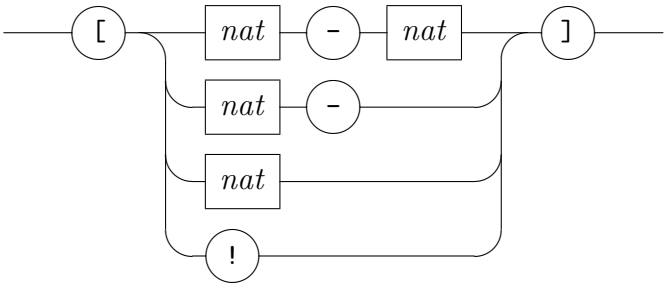
Structural composition "m_1 ; m_2" means that method m_1 is applied with restriction to the first subgoal, then m_2 is applied consecutively with restriction to each subgoal that has newly emerged due to m_1. This is analogous to the tactic combinator `THEN_ALL_NEW` in Isabelle/ML, see also [55]. For example, (*rule r*; *blast*) applies rule r and then solves all new subgoals by *blast*.

Moreover, the explicit goal restriction operator "[n]" exposes only the first n subgoals (which need to exist), with default $n = 1$. For example, the method expression "*simp_all*[3]" simplifies the first three subgoals, while "(*rule r*,

simp_all)[]" simplifies all new goals that emerge from applying rule r to the originally first one.

Improper methods, notably tactic emulations, offer low-level goal addressing as explicit argument to the individual tactic being involved. Here "[!]" refers to all goals, and "[n−]" to all goals starting from n.

goal_spec



6.4.2 Initial and terminal proof steps

proof	:	$proof(prove) \rightarrow proof(state)$
qed	:	$proof(state) \rightarrow proof(state) \mid local_theory \mid theory$
by	:	$proof(prove) \rightarrow proof(state) \mid local_theory \mid theory$
..	:	$proof(prove) \rightarrow proof(state) \mid local_theory \mid theory$
.	:	$proof(prove) \rightarrow proof(state) \mid local_theory \mid theory$
sorry	:	$proof(prove) \rightarrow proof(state) \mid local_theory \mid theory$
standard	:	*method*

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

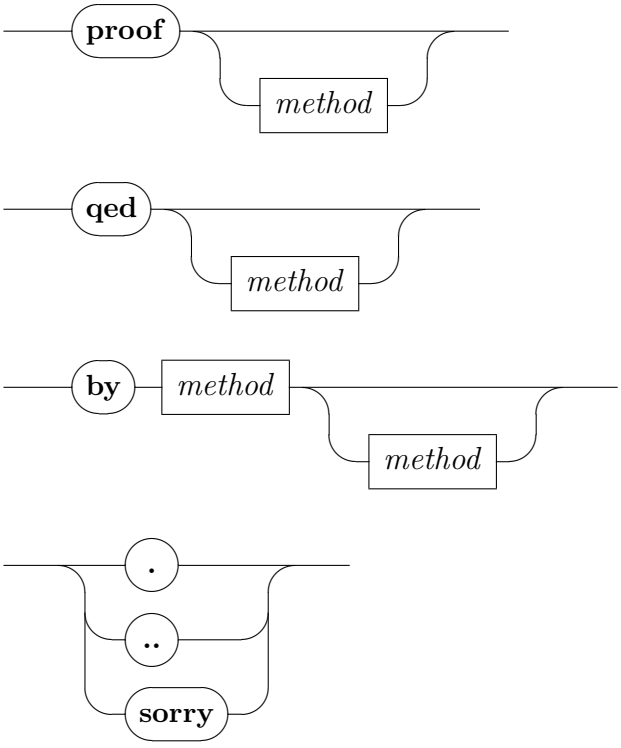
1. An *initial* refinement step **proof** m_1 reduces a newly stated goal to a number of sub-goals that are to be solved later. Facts are passed to m_1 for forward chaining, if so indicated by $proof(chain)$ mode.

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 *standard*, which subsumes at least *rule* or its classical variant *rule*. These methods apply a single standard elimination or introduction rule according to the topmost logical connective 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*, 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[2] 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**.

by m_1 m_2 is a *terminal proof*; it abbreviates **proof** m_1 **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 *standard proof*; it abbreviates **by** *standard*.

"**.**" 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* is enabled. Facts emerging from fake proofs are not the real thing. Internally, the derivation object is tainted by an oracle invocation, which may be inspected via the command **thm_oracles** (§5.15).

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

standard refers to the default refinement step of some Isar language elements (notably **proof** and "**..**"). It is *dynamically scoped*, so the behaviour depends on the application environment.

In Isabelle/Pure, *standard* performs elementary introduction / elimination steps (*rule*), introduction of type classes (*intro_classes*) and locales (*intro_locales*).

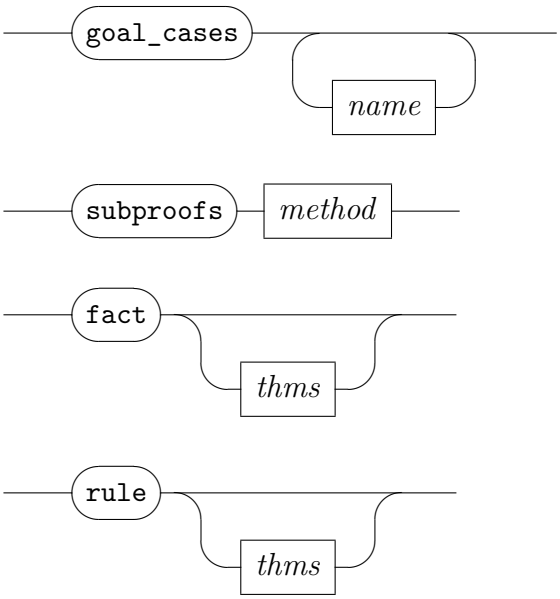
In Isabelle/HOL, *standard* also takes classical rules into account (cf. §9.4).

6.4.3 Fundamental methods and attributes

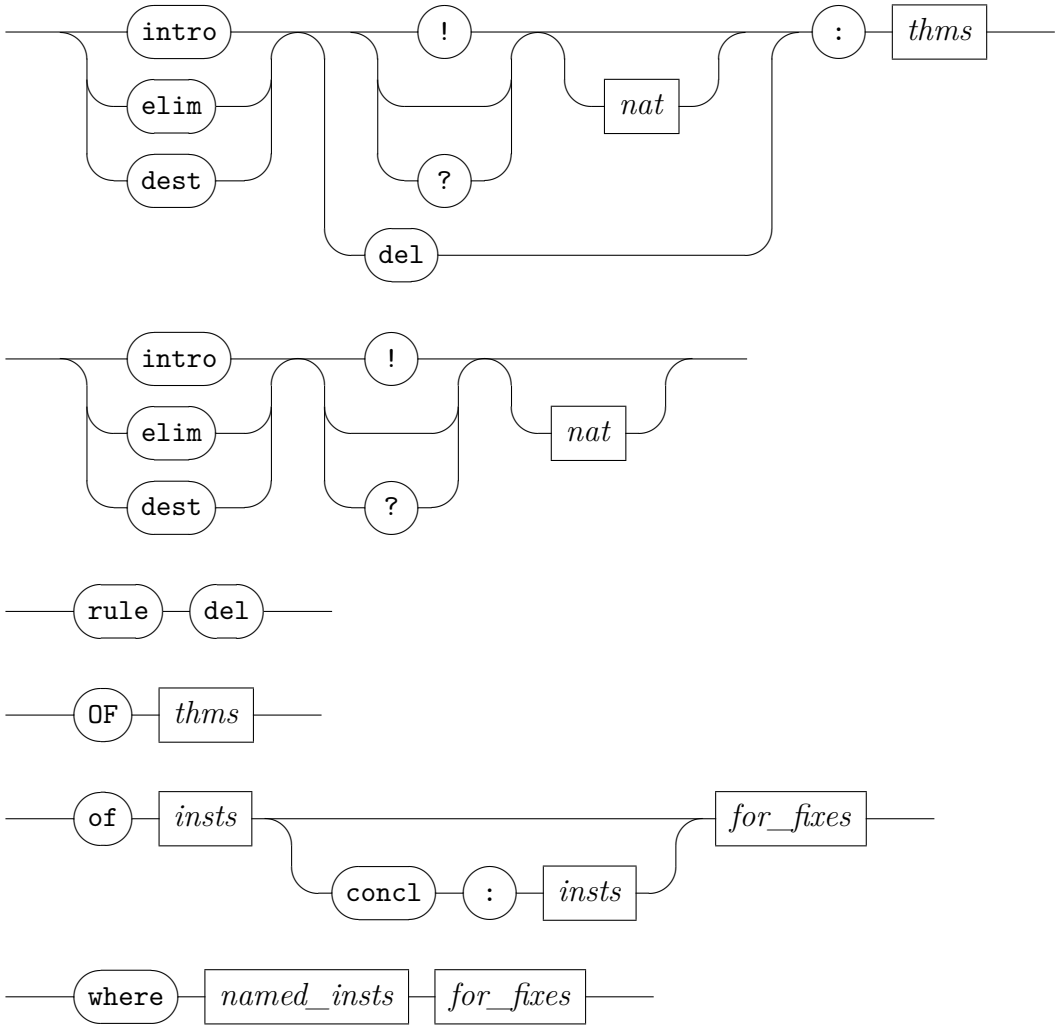
The following proof methods and attributes refer to basic logical operations of Isar. Further methods and attributes are provided by several generic and object-logic specific tools and packages (see chapter 9 and part III).

[2]This includes any additional "strong" assumptions as introduced by **assume**.

print_rules*	:	*context* $\rightarrow$
–	:	*method*
goal_cases	:	*method*
subproofs	:	*method*
fact	:	*method*
assumption	:	*method*
this	:	*method*
rule	:	*method*
intro	:	*attribute*
elim	:	*attribute*
dest	:	*attribute*
rule	:	*attribute*
OF	:	*attribute*
of	:	*attribute*
where	:	*attribute*



rulemod



print_rules prints rules declared via attributes *intro*, *elim*, *dest* of Isabelle/Pure.

See also the analogous **print_claset** command for similar rule declarations of the classical reasoner (§9.4).

"−" (minus) inserts the forward chaining facts as premises into the goal, and nothing else.

Note that command **proof** without any method actually performs a single reduction step using the *rule* method; thus a plain *do-nothing* proof step would be "**proof** −" rather than **proof** alone.

goal_cases $a_1 \ldots a_n$ turns the current subgoals into cases within the context (see also §6.5). The specified case names are used if present; otherwise cases are numbered starting from 1.

Invoking cases in the subsequent proof body via the **case** command will **fix** goal parameters, **assume** goal premises, and **let** variable *?case* refer to the conclusion.

subproofs m applies the method expression m consecutively to each subgoal, constructing individual subproofs internally (analogous to "**show** *goal* **by** m" for each subgoal of the proof state). Search alternatives of m are truncated: the method is forced to be deterministic. This method combinator impacts the internal construction of proof terms: it makes a cascade of let-expressions within the derivation tree and may thus improve scalability.

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. §3.3.9): "**have** φ **by** *fact*" is equivalent to "**note** 'φ'" provided that $\vdash \varphi$ is an instance of some known $\vdash \varphi$ in the proof context.

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 **qed** (§6.4.2) already concludes any remaining sub-goals by assumption, so structured proofs usually need not quote the *assumption* method at all.

this applies all of the current facts directly as rules. Recall that "**.**" (dot) abbreviates "**by** *this*".

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 *rule* without facts is plain introduction, while with facts it becomes elimination.

When no arguments are given, the *rule* method tries to pick appropriate rules automatically, as declared in the current context using the *intro*, *elim*, *dest* attributes (see below). This is included in the standard behaviour of **proof** and "**..**" (double-dot) steps (see §6.4.2).

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 $a_1 \ \ldots \ a_n$ applies some theorem to all of the given rules $a_1, \ldots, a_n$ in canonical right-to-left order, which means that premises stemming from the a_i emerge in parallel in the result, without interfering with each other. In many practical situations, the a_i do not have premises themselves, so *rule* [*OF* $a_1 \ \ldots \ a_n$] 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 $t_1 \ \ldots \ t_n$ performs positional instantiation of term variables. The terms $t_1, \ldots, t_n$ 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.

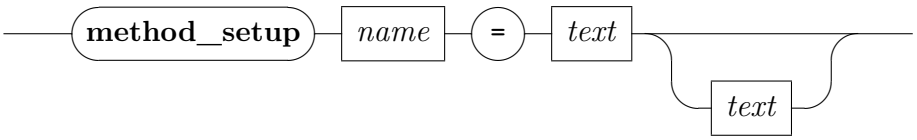
An optional context of local variables **for** $x_1 \ \ldots \ x_m$ may be specified: the instantiated theorem is exported, and these variables become schematic (usually with some shifting of indices).

where $x_1 = t_1$ **and** $\ldots \ x_n = t_n$ 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. *?x*1.3). 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.

An optional context of local variables **for** $x_1 \ \ldots \ x_m$ may be specified as for *of* above.

6.4.4 Defining proof methods

method_setup : *local_theory* → *local_theory*



method_setup *name* = *text description* defines a proof method in the current context. The given *text* has to be an ML expression of type `(Proof.context -> Proof.method) context_parser`, cf. basic parsers defined in structure `Args` and `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 =
  ‹Scan.succeed (K (SIMPLE_METHOD' (fn i: int => no_tac)))›
  "my first method (without any arguments)"

method_setup my_method2 =
  ‹Scan.succeed (fn ctxt: Proof.context =>
    SIMPLE_METHOD' (fn i: int => no_tac))›
  "my second method (with context)"

method_setup my_method3 =
  ‹Attrib.thms >> (fn thms: thm list => fn ctxt: Proof.context =>
    SIMPLE_METHOD' (fn i: int => no_tac))›
  "my third method (with theorem arguments and context)"
```

6.5 Proof by cases and induction

6.5.1 Rule contexts

case	:	$proof(state) \to proof(state)$
print_cases*	:	$context \to$
case_names	:	*attribute*
case_conclusion	:	*attribute*
params	:	*attribute*
consumes	:	*attribute*

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.

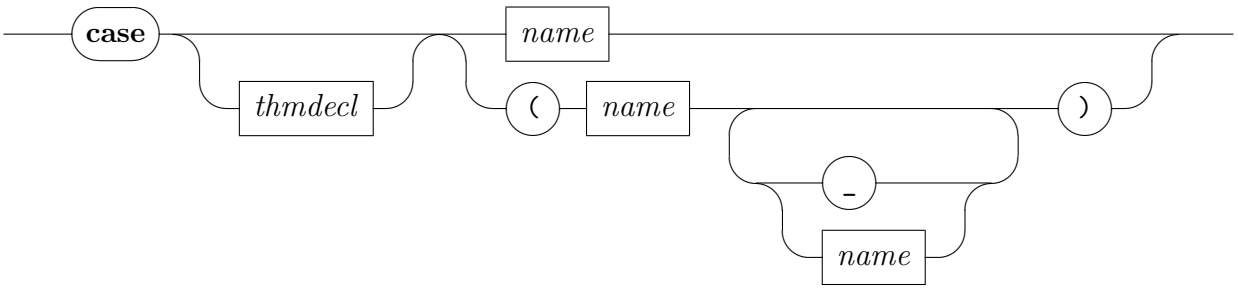
The **case** command provides a shorthand to refer to a local context symbolically: certain proof methods provide an environment of named "cases" of the form c: $x_1, \ldots, x_m$, $\varphi_1, \ldots, \varphi_n$; the effect of "**case** c" is then equivalent to "**fix** $x_1 \ldots x_m$ **assume** c: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

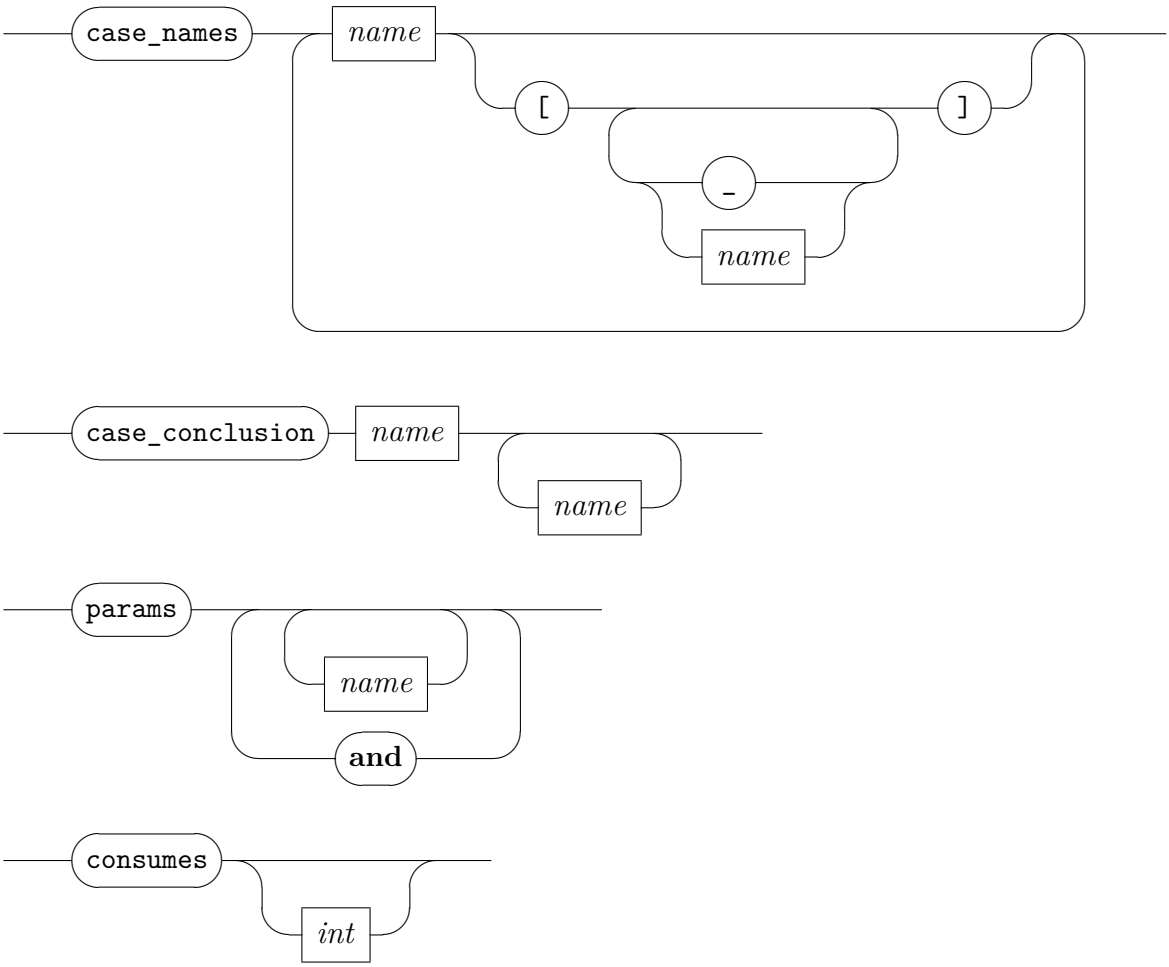
By default, the "terminology" $x_1, \ldots, x_m$ of a case value is marked as hidden, i.e. there is no way to refer to such parameters in the subsequent proof text. After all, original rule parameters stem from somewhere outside of the current proof text. By using the explicit form "**case** $(c\ y_1 \ldots\ y_m)$" 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 *a*: (*c* x_1 ... x_m) invokes a named local context *c*: x_1, ..., x_m, φ_1, ..., φ_m, as provided by an appropriate proof method (such as *cases* and *induct*). The command "**case** *a*: (*c* x_1 ... x_m)" abbreviates "**fix** x_1 ... x_m **assume** *a.c*: φ_1 ... φ_n". Each local fact is qualified by the prefix *a*, and all such facts are collectively bound to the name *a*.

The fact name is specification *a* is optional, the default is to re-use *c*. So **case** (*c* x_1 ... x_m) is the same as **case** *c*: (*c* x_1 ... x_m).

print_cases prints all local contexts of the current state, using Isar proof language notation.

case_names c_1 ... c_k declares names for the local contexts of premises of a theorem; c_1, ..., 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 ... h_n are the names of the hypotheses in case c_i from left to right.

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

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

params p_1 ... p_m **and** ... q_1 ... q_n renames the innermost parameters of premises 1, ..., 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. §11.1). Rules without any *consumes* declaration given are treated as if *consumes* 0 had been specified.

A negative n is interpreted relatively to the total number of premises of the rule in the target context. Thus its absolute value specifies the remaining number of premises, after subtracting the prefix of major premises as indicated above. This form of declaration has the technical advantage of being stable under more morphisms, notably those that export the result from a nested **context** with additional assumptions.

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.

6.5.2 Proof methods

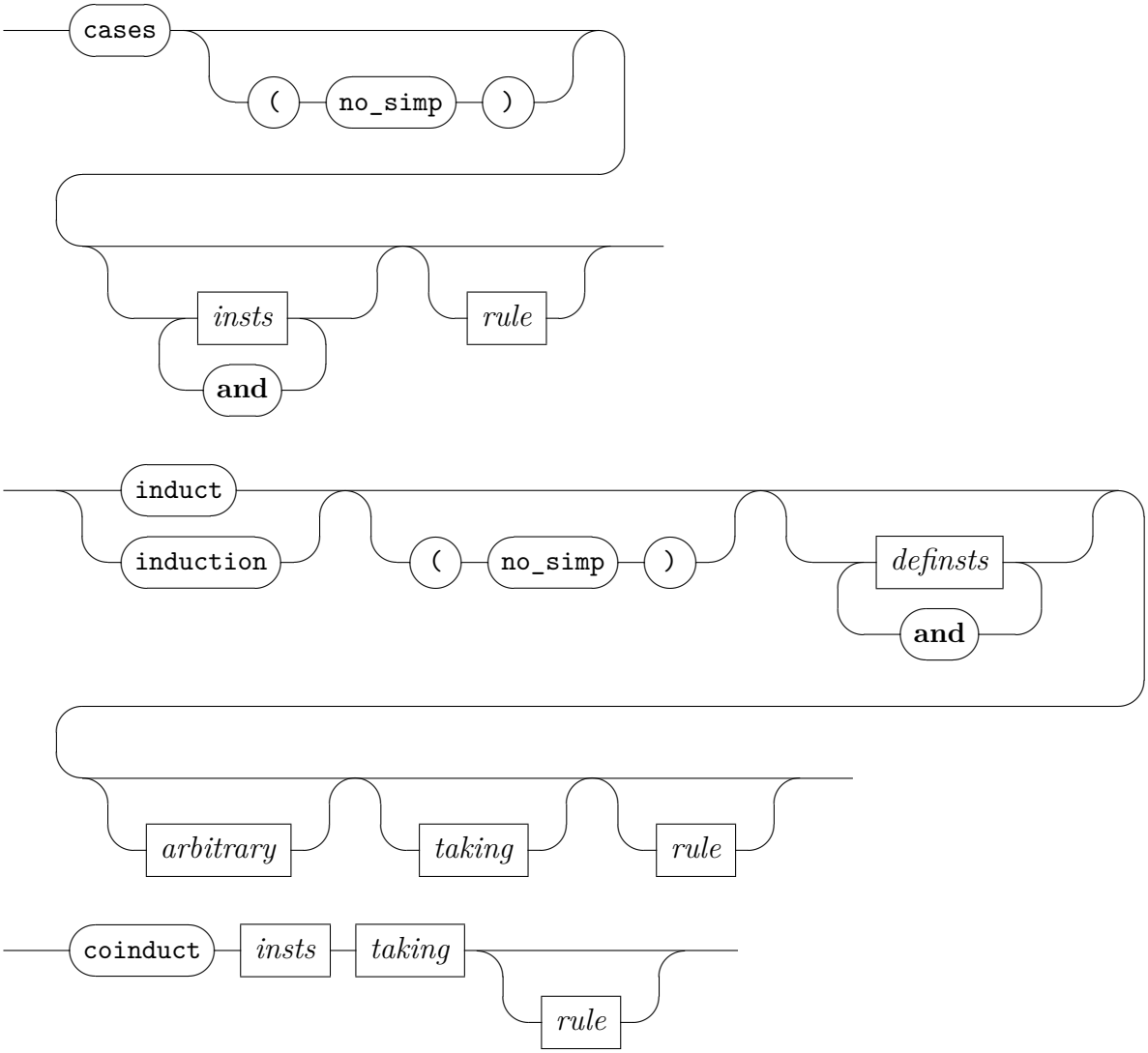
cases : *method*
induct : *method*
induction : *method*
coinduct : *method*

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

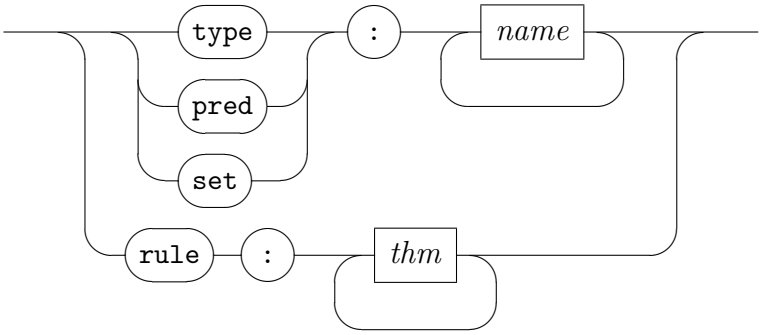
instantiated in a casual manner. Furthermore, these methods provide named local contexts that may be invoked via the **case** proof command within the subsequent proof text. This accommodates compact proof texts even when reasoning about large specifications.

The *induct* method also provides some infrastructure to work with structured 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.

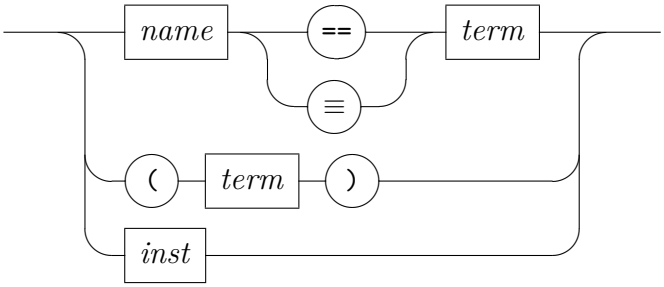
Method *induction* differs from *induct* only in the names of the facts in the local context invoked by the **case** command.



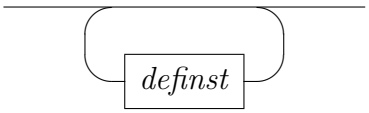
rule



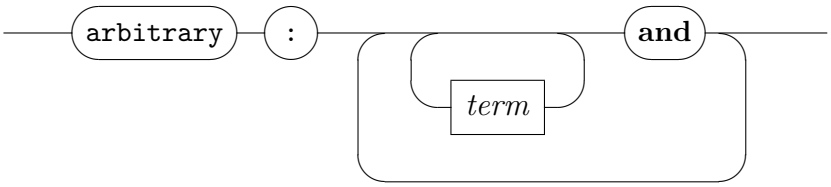
definst



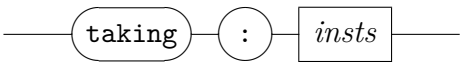
definsts



arbitrary



taking



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:

facts		arguments	rule
$\vdash R$	*cases*		implicit rule R
	cases		classical case split
	cases	t	datatype exhaustion (type of t)
$\vdash A\ t$	*cases*	...	inductive predicate/set elimination (of A)
...	*cases*	... *rule*: R	explicit rule R

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:

facts		arguments	rule
	induct	$P\ x$	datatype induction (type of x)
$\vdash A\ x$	*induct*	...	predicate/set induction (of A)
...	*induct*	... *rule*: R	explicit rule R

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 generalized (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 (*no_simp*) option can be used to disable pre-simplification. Additional rules to be used in pre-simplification can be declared using the *induct_simp* attribute.

The optional "*arbitrary*: $x_1 \ldots x_m$" specification generalizes variables $x_1, \ldots, x_m$ of the original goal before applying induction. It is possible to separate variables by "*and*" to generalize in goals other than 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 "*taking*: $t_1 \ldots t_n$" specification provides additional instantiations of a prefix of pending variables in the rule. Such schematic induction rules rarely occur in practice, though.

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

goal		arguments	rule
	coinduct	x	type coinduction (type of x)
$A\ x$	*coinduct*	...	predicate/set coinduction (of A)
...	*coinduct*	... *rule*: R	explicit rule R

Coinduction is the dual of induction. Induction essentially eliminates $A\ x$ towards a generic result $P\ x$, while coinduction introduces $A\ x$ starting with $B\ x$, for a suitable "bisimulation" B. 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 *suffix* of variables occurring in the rule's major premise, or conclusion if unavailable. An additional "*taking*: $t_1 \ldots t_n$" 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 *induct* and *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 *?case* for the conclusion will be provided with each case, provided that term is fully specified.

The **print_cases** command prints all named cases present in the current proof state.

Despite the additional infrastructure, both *cases* and *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 $\bigwedge x_1 \ldots x_m.\ \varphi_1 \Longrightarrow \ldots\ \varphi_n \Longrightarrow \ldots$ reappears unchanged after the case split.

The *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.

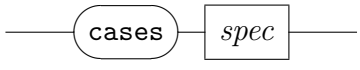
In *induct* proofs, local assumptions introduced by cases are split into two different kinds: *hyps* stemming from the rule and *prems* from the goal statement. This is reflected in the extracted cases accordingly, so invoking "**case** *c*" will provide separate facts *c.hyps* and *c.prems*, as well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

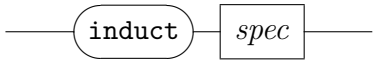
In *induction* proofs, local assumptions introduced by cases are split into three different kinds: *IH*, the induction hypotheses, *hyps*, the remaining hypotheses stemming from the rule, and *prems*, the assumptions from the goal statement. The names are *c.IH*, *c.hyps* and *c.prems*, as above.

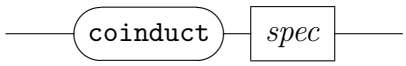
Facts presented to either method are consumed according to the number of "major premises" of the rule involved, which is usually 0 for plain cases and induction rules of datatypes etc. and 1 for rules of inductive predicates or sets and the like. The remaining facts are inserted into the goal verbatim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

6.5.3 Declaring rules

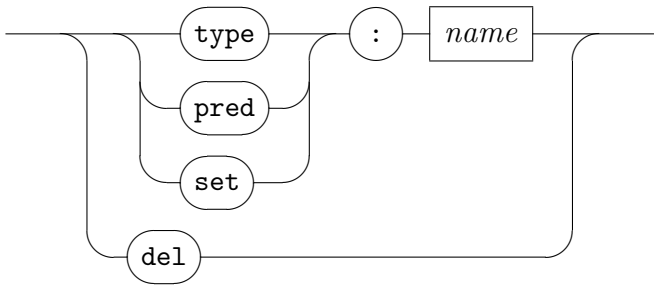
print_induct_rules* : *context* $\to$
cases : *attribute*
induct : *attribute*
coinduct : *attribute*







spec



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.

6.6 Generalized elimination and case splitting

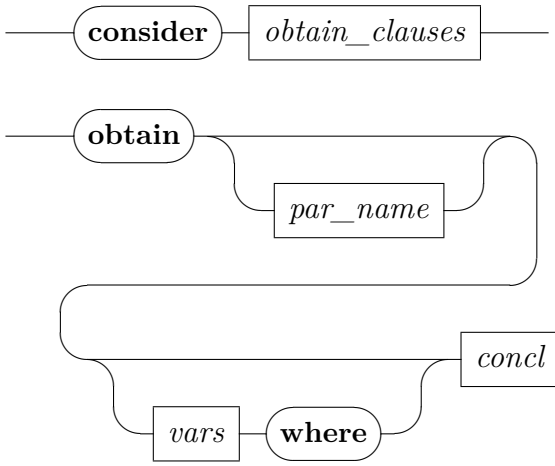
consider : *proof*(*state*) | *proof*(*chain*) → *proof*(*prove*)
obtain : *proof*(*state*) | *proof*(*chain*) → *proof*(*prove*)

Generalized elimination means that hypothetical parameters and premises may be introduced in the current context, potentially with a split into cases. This works by virtue of a locally proven rule that establishes the soundness

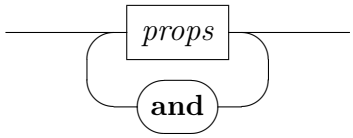
of this temporary context extension. As representative examples, one may think of standard rules from Isabelle/HOL like this:

$\exists x.\ B\ x \Longrightarrow (\bigwedge x.\ B\ x \Longrightarrow \mathit{thesis}) \Longrightarrow \mathit{thesis}$
$A \wedge B \Longrightarrow (A \Longrightarrow B \Longrightarrow \mathit{thesis}) \Longrightarrow \mathit{thesis}$
$A \vee B \Longrightarrow (A \Longrightarrow \mathit{thesis}) \Longrightarrow (B \Longrightarrow \mathit{thesis}) \Longrightarrow \mathit{thesis}$

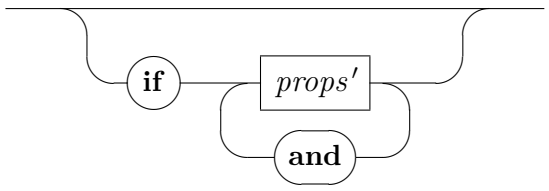
In general, these particular rules and connectives need to get involved at all: this concept works directly in Isabelle/Pure via Isar commands defined below. In particular, the logic of elimination and case splitting is delegated to an Isar proof, which often involves automated tools.



concl



prems



consider $(a)\ \overline{x}$ **where** $\overline{A}\ \overline{x}$ | $(b)\ \overline{y}$ **where** $\overline{B}\ \overline{y}$ | ... states a rule for case splitting into separate subgoals, such that each case involves new parameters and premises. After the proof is finished, the resulting rule

may be used directly with the *cases* proof method (§6.5), in order to perform actual case-splitting of the proof text via **case** and **next** as usual.

Optional names in round parentheses refer to case names: in the proof of the rule this is a fact name, in the resulting rule it is used as annotation with the *case_names* attribute.

Formally, the command **consider** is defined as derived Isar language element as follows:

consider (a) $\overline{x}$ **where** $\overline{A}$ $\overline{x}$ | (b) $\overline{y}$ **where** $\overline{B}$ $\overline{y}$ | ... $\equiv$
  have [*case_names a b ...*]: *thesis*
    if *a* [*Pure.intro?*]: $\bigwedge \overline{x}.\ \overline{A}\ \overline{x} \Longrightarrow$ *thesis*
    and *b* [*Pure.intro?*]: $\bigwedge \overline{y}.\ \overline{B}\ \overline{y} \Longrightarrow$ *thesis*
    and ...
    for *thesis*
    apply (*insert a b ...*)

See also §6.2.4 for **obtains** in toplevel goal statements, as well as **print_statement** to print existing rules in a similar format.

obtain $\overline{x}$ **where** $\overline{A}$ $\overline{x}$ states a generalized elimination rule with exactly one case. After the proof is finished, it is activated for the subsequent proof text: the context is augmented via **fix** $\overline{x}$ **assume** $\overline{A}$ $\overline{x}$, with special provisions to export later results by discharging these assumptions again.

Note that according to the parameter scopes within the elimination rule, results *must not* refer to hypothetical parameters; otherwise the export will fail! This restriction conforms to the usual manner of existential reasoning in Natural Deduction.

Formally, the command **obtain** is defined as derived Isar language element as follows, using an instrumented variant of **assume**:

obtain $\overline{x}$ **where** a: $\overline{A}\ \overline{x}$ $\langle proof \rangle \equiv$

have *thesis*
if *that* [*Pure.intro?*]: $\bigwedge \overline{x}.\ \overline{A}\ \overline{x} \Longrightarrow thesis$
for *thesis*
apply (*insert that*)
$\langle proof \rangle$
fix $\overline{x}$ **assume*** a: $\overline{A}\ \overline{x}$

In the proof of **consider** and **obtain** the local premises are always bound to the fact name *that*, according to structured Isar statements involving **if** (§6.2.4).

Facts that are established by **obtain** cannot be polymorphic: any type-variables occurring here are fixed in the present context. This is a natural consequence of the role of **fix** and **assume** in this construct.

Chapter 7

Proof scripts

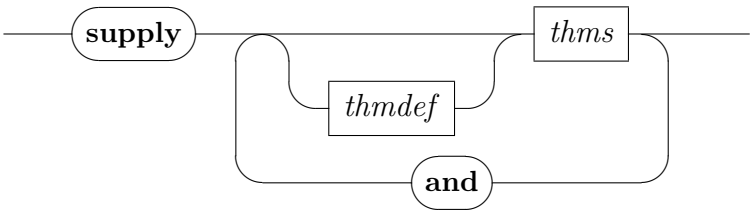
Interactive theorem proving is traditionally associated with "proof scripts", but Isabelle/Isar is centered around structured *proof documents* instead (see also chapter 6).

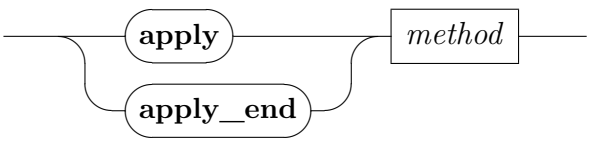
Nonetheless, it is possible to emulate proof scripts by sequential refinements of a proof state in backwards mode, notably with the **apply** command (see §7.1).

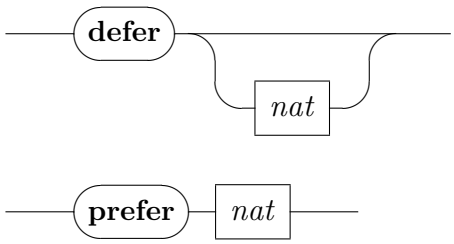
There are also various proof methods that allow to refer to implicit goal state information that is not accessible to structured Isar proofs (see §7.3). Note that the **subgoal** (§7.2) command usually eliminates the need for implicit goal state references.

7.1 Commands for step-wise refinement

supply* : $proof(prove) \to proof(prove)$
apply* : $proof(prove) \to proof(prove)$
apply_end* : $proof(state) \to proof(state)$
done* : $proof(prove) \to proof(state) \mid local_theory \mid theory$
defer* : $proof \to proof$
prefer* : $proof \to proof$
back* : $proof \to proof$







supply supports fact definitions during goal refinement: it is similar to **note**, but it operates in backwards mode and does not have any impact on chained facts.

apply *m* applies proof method *m* in initial position, but unlike **proof** it retains "*proof*(*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 **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 **qed**). Thus the proof method may not refer to any assumptions introduced in the current body, for example.

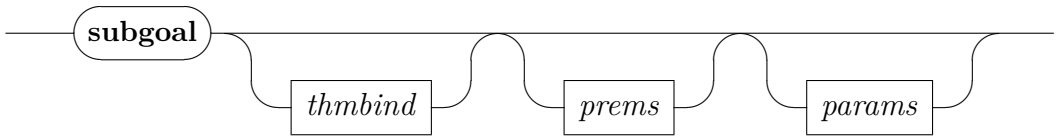
done completes a proof script, provided that the current goal state is solved completely. Note that actual structured proof commands (e.g. "**.**" or **sorry**) may be used to conclude proof scripts as well.

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.

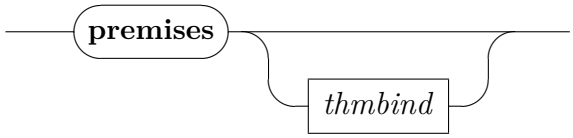
back does back-tracking over the result sequence of the latest proof command. Any proof command may return multiple results, and this command explores the possibilities step-by-step. It is mainly useful for experimentation and interactive exploration, and should be avoided in finished proofs.

7.2 Explicit subgoal structure

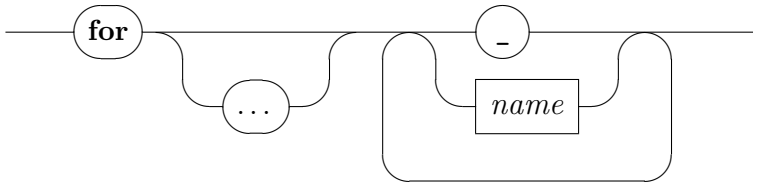
subgoal* : *proof* $\rightarrow$ *proof*



prems



params



subgoal allows to impose some structure on backward refinements, to avoid proof scripts degenerating into long of **apply** sequences.

The current goal state, which is essentially a hidden part of the Isar/VM configuration, is turned into a proof context and remaining conclusion. This corresponds to **fix** / **assume** / **show** in structured proofs, but the text of the parameters, premises and conclusion is not given explicitly.

Goal parameters may be specified separately, in order to allow referring to them in the proof body: "**subgoal for** *x y z*" names a *prefix*, and "**subgoal for** ... *x y z*" names a *suffix* of goal parameters. The latter uses a literal `\<dots>` symbol as notation. Parameter positions may be skipped via dummies (underscore). Unspecified names remain internal, and thus inaccessible in the proof text.

"**subgoal premises** *prems*" indicates that goal premises should be turned into assumptions of the context (otherwise the remaining conclusion is a Pure implication). The fact name and attributes are optional; the particular name "*prems*" is a common convention for the premises of an arbitrary goal context in proof scripts.

"**subgoal** *result*" indicates a fact name for the result of a proven subgoal. Thus it may be re-used in further reasoning, similar to the result of **show** in structured Isar proofs.

Here are some abstract examples:

lemma $\bigwedge x\ y\ z.\ A\ x \Longrightarrow B\ y \Longrightarrow C\ z$
and $\bigwedge u\ v.\ X\ u \Longrightarrow Y\ v$
subgoal ⟨*proof*⟩
subgoal ⟨*proof*⟩
done

lemma $\bigwedge x\ y\ z.\ A\ x \Longrightarrow B\ y \Longrightarrow C\ z$
and $\bigwedge u\ v.\ X\ u \Longrightarrow Y\ v$
subgoal for $x\ y\ z$ ⟨*proof*⟩
subgoal for $u\ v$ ⟨*proof*⟩
done

lemma $\bigwedge x\ y\ z.\ A\ x \Longrightarrow B\ y \Longrightarrow C\ z$
and $\bigwedge u\ v.\ X\ u \Longrightarrow Y\ v$
subgoal premises for $x\ y\ z$
using ‹$A\ x$› ‹$B\ y$›
⟨*proof*⟩
subgoal premises for $u\ v$
using ‹$X\ u$›
⟨*proof*⟩
done

lemma $\bigwedge x\ y\ z.\ A\ x \Longrightarrow B\ y \Longrightarrow C\ z$
and $\bigwedge u\ v.\ X\ u \Longrightarrow Y\ v$
subgoal r **premises** *prems* **for** $x\ y\ z$
proof −
have $A\ x$ **by** (*fact prems*)
moreover have $B\ y$ **by** (*fact prems*)
ultimately show *?thesis* ⟨*proof*⟩
qed
subgoal premises *prems* **for** $u\ v$
proof −
have $\bigwedge x\ y\ z.\ A\ x \Longrightarrow B\ y \Longrightarrow C\ z$ **by** (*fact r*)
moreover
have $X\ u$ **by** (*fact prems*)
ultimately show *?thesis* ⟨*proof*⟩
qed
done

lemma $\bigwedge x\ y\ z.\ A\ x \Longrightarrow B\ y \Longrightarrow C\ z$
subgoal premises *prems* **for** ... z
proof −
from *prems* **show** $C\ z$ ⟨*proof*⟩
qed
done

7.3 Tactics: improper proof methods

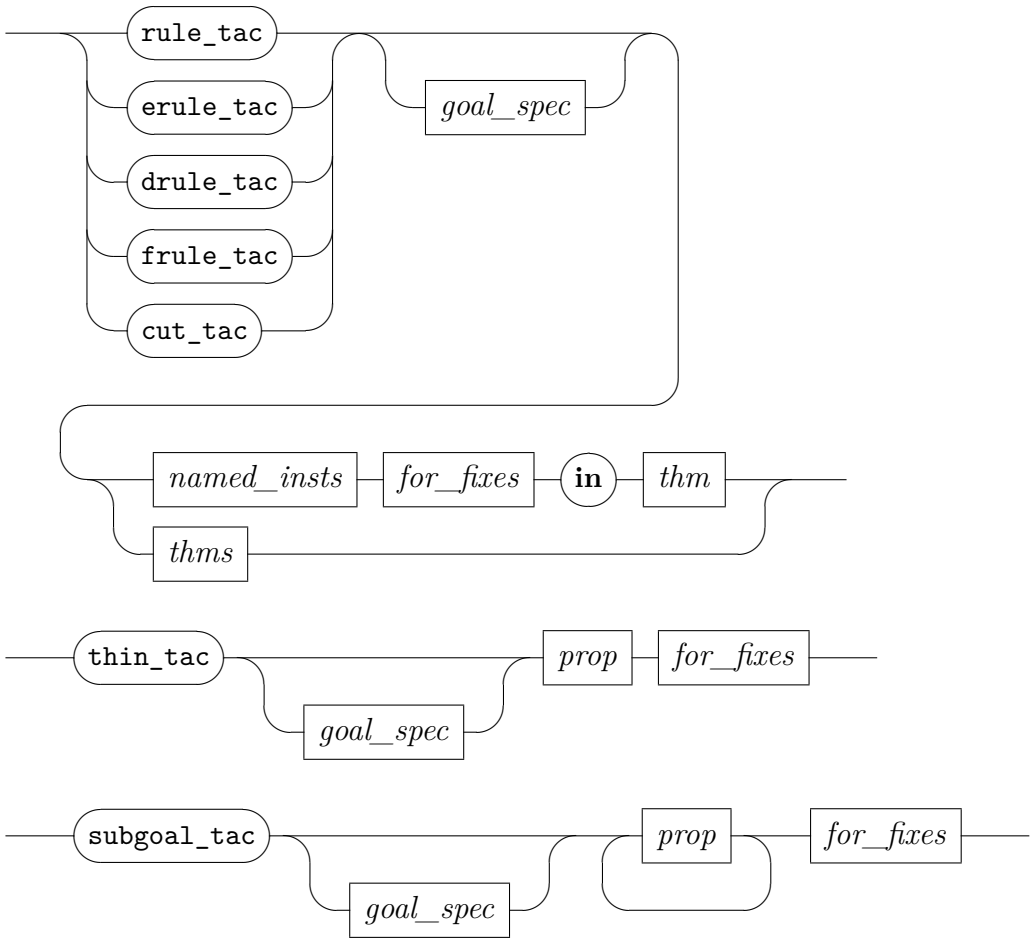
The following improper proof methods emulate traditional tactics. These admit direct access to the goal state, which is normally considered harmful! In particular, this may involve both numbered goal addressing (default 1), and dynamic instantiation within the scope of some subgoal.

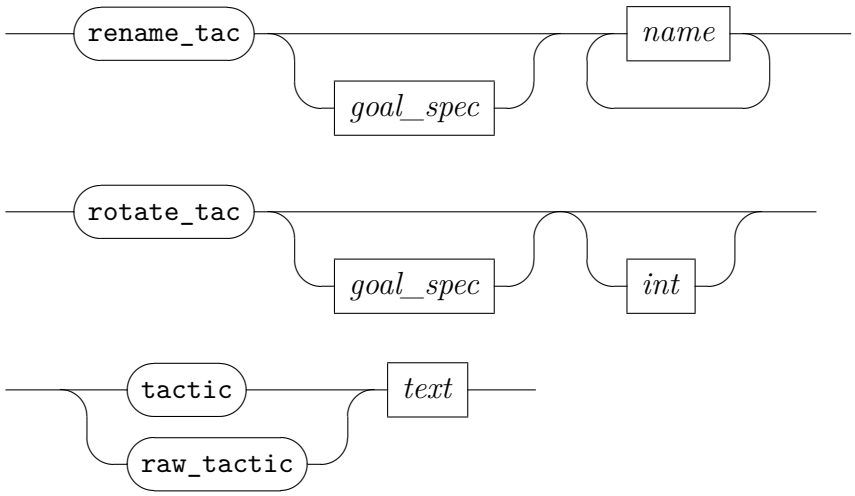
! Dynamic instantiations refer to universally quantified parameters of a subgoal (the dynamic context) rather than fixed variables and term abbreviations of a (static) Isar context.

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 *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 *where* (see §6.4.3).

$$
\begin{array}{rcl}
\mathit{rule_tac}^* & : & \mathit{method} \\
\mathit{erule_tac}^* & : & \mathit{method} \\
\mathit{drule_tac}^* & : & \mathit{method} \\
\mathit{frule_tac}^* & : & \mathit{method} \\
\mathit{cut_tac}^* & : & \mathit{method} \\
\mathit{thin_tac}^* & : & \mathit{method} \\
\mathit{subgoal_tac}^* & : & \mathit{method} \\
\mathit{rename_tac}^* & : & \mathit{method} \\
\mathit{rotate_tac}^* & : & \mathit{method} \\
\mathit{tactic}^* & : & \mathit{method} \\
\mathit{raw_tactic}^* & : & \mathit{method}
\end{array}
$$





rule_tac etc. do resolution of rules with explicit instantiation. This works the same way as the ML tactics `Rule_Insts.res_inst_tac` etc. (see [55]).

Multiple rules may be only given if there is no instantiation; then *rule_tac* is the same as `resolve_tac` in ML (see [55]).

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 *insert* which inserts closed rule statements.

thin_tac φ deletes the specified premise from a subgoal. Note that φ 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.

Chapter 8

Inner syntax — the term language

The inner syntax of Isabelle provides concrete notation for the main entities of the logical framework, notably λ-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 `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 (§8.2), which may be usually given as part of the original declaration or via explicit notation commands later on (§8.3). This already covers many needs of concrete syntax without having to understand the full complexity of inner syntax layers.

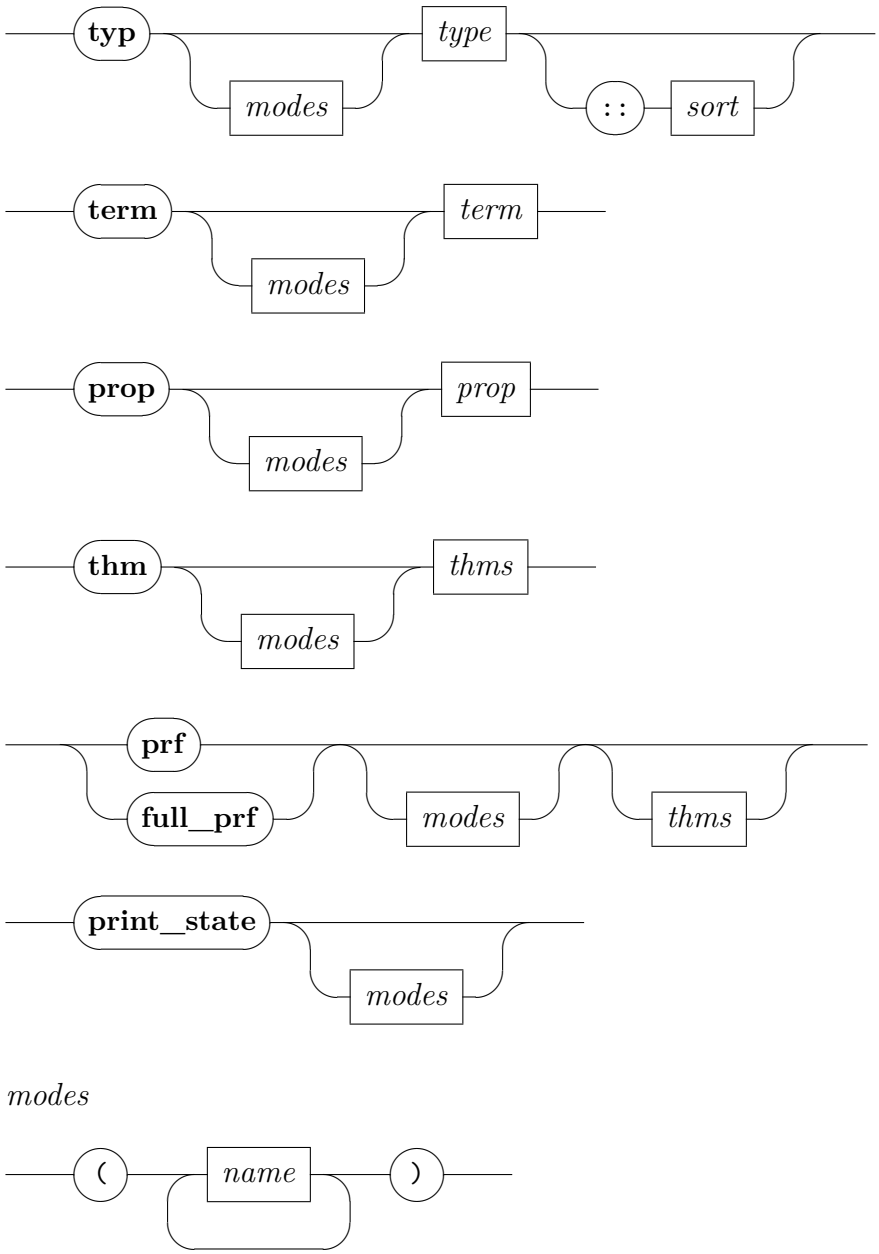
Further details of the syntax engine involves the classical distinction of lexical language versus context-free grammar (see §8.4), and various mechanisms for *syntax transformations* (see §8.5).

8.1 Printing logical entities

8.1.1 Diagnostic commands

typ* : *context* $\rightarrow$
term* : *context* $\rightarrow$
prop* : *context* $\rightarrow$
thm* : *context* $\rightarrow$
prf* : *context* $\rightarrow$
full_prf* : *context* $\rightarrow$
print_state* : *any* $\rightarrow$

These diagnostic commands assist interactive development by printing internal logical entities in a human-readable fashion.



modes

typ τ reads and prints a type expression according to the current context.

typ τ :: s uses type-inference to determine the most general way to make τ conform to sort s. For concrete τ this checks if the type belongs to that sort. Dummy type parameters "_" (underscore) are assigned to fresh type variables with most general sorts, according the the principles of type-inference.

term t and **prop** φ 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 an underlying logic image with proof terms enabled, e.g. *HOL−Proofs*.

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.

print_state prints the current proof state (if present), including current facts and goals.

The diagnostic commands above accept an optional list of *modes*, which is appended to the current print mode; see also §8.1.3. Thus the output behavior may be modified according particular print mode features. For example, **thm** `("") symmetric` prints a theorem without any special markup, bypassing the print mode setup of the Prover IDE.

8.1.2 Details of printed content

show_markup	:	*attribute*	
show_consts_markup	:	*attribute*	default *true*
show_types	:	*attribute*	default *false*
show_sorts	:	*attribute*	default *false*
show_consts	:	*attribute*	default *false*
show_abbrevs	:	*attribute*	default *true*
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_markup controls direct inlining of markup into the printed representation of formal entities — notably type and sort constraints. This enables Prover IDE users to retrieve that information via tooltips or popups while hovering with the mouse over the output window, for example. Consequently, this option is enabled by default for Isabelle/jEdit.

show_consts_markup controls printing of type constrains for term constants; this requires *show_markup*.

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. In Isabelle/jEdit, manual setting of these options is normally not required thanks to *show_markup* above.

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_abbrevs controls folding of constant abbreviations.

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 η-contracted printing of terms.

The η-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 η-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 η-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 η-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 printed.

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).

8.1.3 Alternative print modes

```
print_mode_value: unit -> string list
Print_Mode.with_modes: string list -> ('a -> 'b) -> 'a -> 'b
```

The *print mode* facility allows to modify various operations for printing. Commands like **typ**, **term**, **thm** (see §8.1.1) take additional print modes as optional argument. The underlying ML operations are as follows.

`print_mode_value ()` 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).

`Print_Mode.with_modes` *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 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:

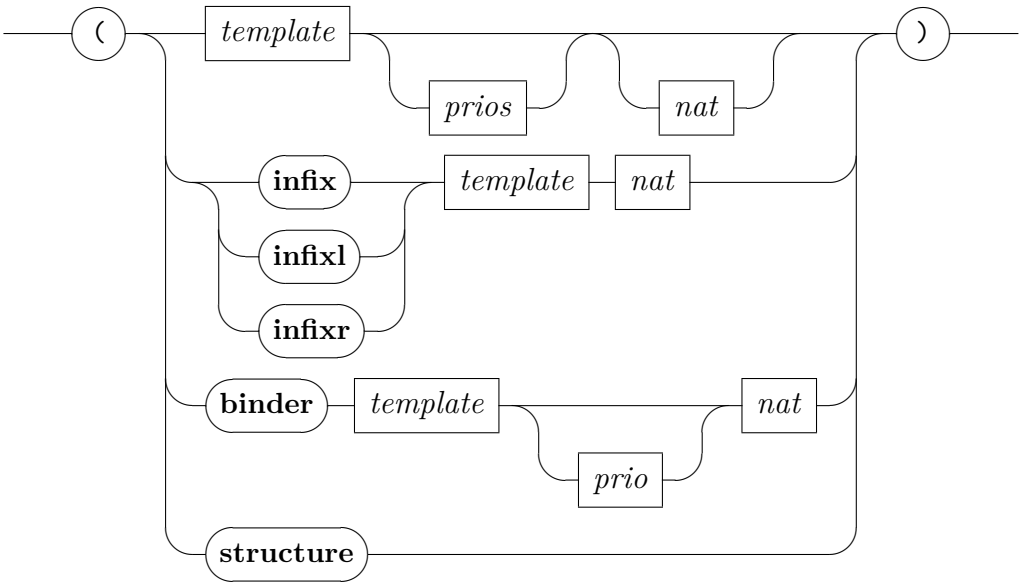
- `""` (the empty string): default mode; implicitly active as last element in the list of modes.
- `input`: dummy print mode that is never active; may be used to specify notation that is only available for input.
- `internal` dummy print mode that is never active; used internally in Isabelle/Pure.
- `ASCII`: prefer ASCII art over mathematical symbols.
- `latex`: additional mode that is active in LaTeX document preparation of Isabelle theory sources; allows to provide alternative output notation.

8.2 Mixfix annotations

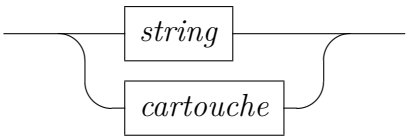
Mixfix annotations specify concrete *inner syntax* of Isabelle types and terms. Locally fixed parameters in toplevel theorem statements, locale and class specifications also admit mixfix annotations in a fairly uniform manner. A mixfix annotation describes the concrete syntax, the translation to abstract 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 [17]. It allows to specify any context-free priority grammar, which is more general than the fixity declarations of ML and Prolog.

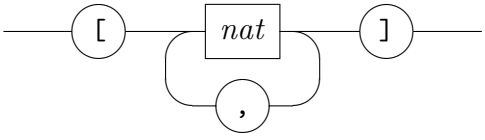
mixfix



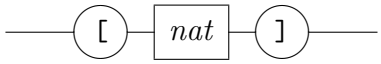
template



prios



prio



The mixfix *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 §5.7). Only locally fixed variables may be declared as **structure**.

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.

8.2.1 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 (*mixfix* $[p_1, \ldots, p_n]\ p$), where *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 τ_i (with priority p_i), and the result category is determined from τ (with priority p). Priority specifications are optional, with default 0 for arguments and 1000 for the result.[1]

Since τ 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 delimiter items of the following form:

1. a control symbol followed by a cartouche

[1] Omitting priorities is prone to syntactic ambiguities unless the delimiter tokens determine fully bracketed notation, as in *if* _ *then* _ *else* _ *fi*.

2. a single symbol, excluding the following special characters:

’	single quote
_	underscore
ı	index symbol
(	open parenthesis
)	close parenthesis
/	slash
‹ ›	cartouche delimiters

’ 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.

ı 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 natural number specifies the block indentation, i.e. how much spaces to add when a line break occurs within the block. The default indentation is 0.

(‹*properties*› opens a pretty printing block, with properties specified within the given text cartouche. The syntax and semantics of the category *mixfix_properties* is described below.

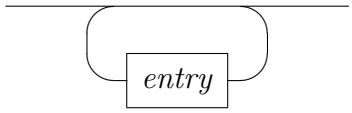
) closes a pretty printing block.

// forces a line break.

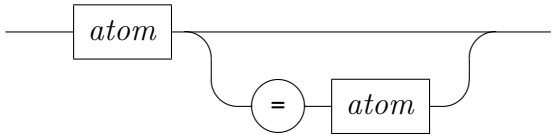
/*s* allows a line break. Here *s* stands for the string of spaces (zero or more) right after the slash. These spaces are printed if the break is *not* taken.

Block properties allow more control over the details of pretty-printed output. The concrete syntax is defined as follows.

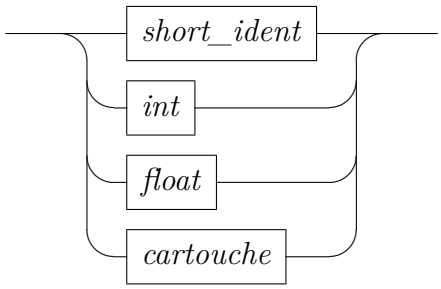
mixfix_properties



entry



atom



Each *entry* is a name–value pair, but the latter is optional. If the value is omitted, the default depends on its type (Boolean: `true`, number: `1`, otherwise the empty string). The following standard block properties are supported:

- *indent* (natural number): the block indentation — the same as for the simple syntax without block properties.
- *open_block* (Boolean): this block has no impact on formatting, but it may carry markup information.
- *consistent* (Boolean): this block has consistent breaks (if one break is taken, all breaks are taken).
- *unbreakable* (Boolean): all possible breaks of the block are disabled (turned into spaces).
- *markup* (string): the optional name of the markup node. If this is provided, all remaining properties are turned into its XML attributes. This allows to specify free-form PIDE markup, e.g. for specialized output.

- *notation* (cartouche): a semi-formal description of the notation that is surrounded by the block parentheses. The cartouche consists of multiple words (separated by white-space). The first word specifies the *kind* of notation as follows:

  - *mixfix*: general mixfix notation, with delimiters surrounding its arguments.
  - *prefix*: notation with delimiter before its argument.
  - *postfix*: notation with delimiter after its argument.
  - *infix*: notation with delimiter between its arguments (automatically inserted for **infix** annotations, see §8.2.2).
  - *binder*: notation that binds variables within its body argument (automatically inserted for **binder** annotations, see §8.2.3).
  - *literal*: notation for literal values, such as string or number.
  - *type_application*: application of a type constructor to its arguments.
  - *application*: λ-term application.
  - *abstraction*: λ-term abstraction.
  - *judgment*: judgment form of the object-logic (usually without delimiters).

  Plenty of examples may be found in the Isabelle sources by searching for "`notation=`".

Note that the general idea of pretty printing with blocks and breaks is described in [47]; it goes back to [41].

8.2.2 Infixes

Infix operators are specified by convenient short forms that abbreviate general mixfix annotations as follows:

(infix "*sy*" *p*)	$\rightsquigarrow$	("(_ *sy*/ _)" [*p* + 1, *p* + 1] *p*)
(infixl "*sy*" *p*)	$\rightsquigarrow$	("(_ *sy*/ _)" [*p*, *p* + 1] *p*)
(infixr "*sy*" *p*)	$\rightsquigarrow$	("(_ *sy*/ _)" [*p* + 1, *p*] *p*)

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 (*sy*) is introduced in addition. Thus any infix operator may be written in prefix form (as in Haskell), independently of the number of arguments.

8.2.3 Binders

A *binder* is a variable-binding construct such as a quantifier. The idea to formalize $\forall x.\ b$ as $All\ (\lambda x.\ b)$ for $All :: ('a \Rightarrow bool) \Rightarrow bool$ already goes back to [14]. Isabelle declarations of certain higher-order operators may be annotated with **binder** annotations as follows:

$$c :: "(\tau_1 \Rightarrow \tau_2) \Rightarrow \tau_3" \quad \textbf{(binder "}sy\textbf{" [}p\textbf{] }q\textbf{)}$$

This introduces concrete binder syntax *sy* x. b, where x is a bound variable of type τ_1, the body b has type τ_2 and the whole term has type τ_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:

$$c_binder :: "idts \Rightarrow \tau_2 \Rightarrow \tau_3" \quad \textbf{("(3}sy\textbf{_./ _)" [0, }p\textbf{] }q\textbf{)}$$

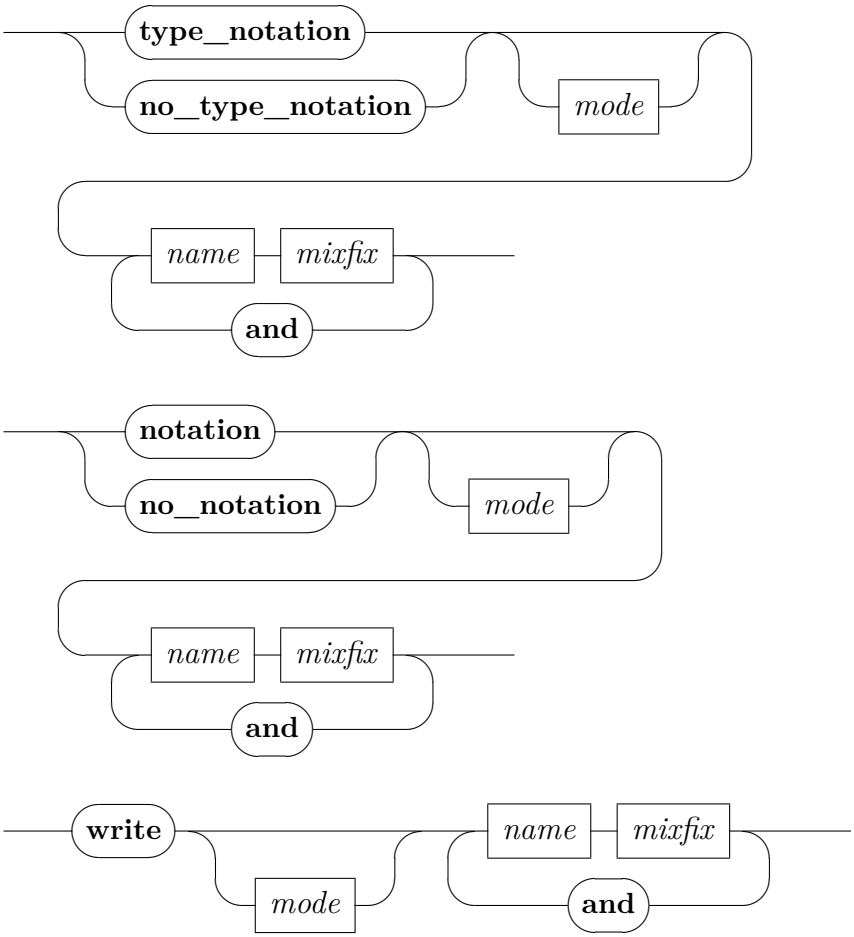
Here *idts* is the nonterminal symbol for a list of identifiers with optional type constraints (see also §8.4.3). The mixfix template "(3*sy*_./ _)" defines argument positions for the bound identifiers and the body, separated by a dot with optional line break; the entire phrase is a pretty printing block 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 $c_binder\ x_1\ \ldots\ x_n\ b$ into iterated application $c\ (\lambda x_1.\ \ldots\ c\ (\lambda x_n.\ b)\ldots)$. This works in both directions, for parsing and printing.

8.3 Explicit notation

type_notation	:	*local_theory* → *local_theory*
no_type_notation	:	*local_theory* → *local_theory*
notation	:	*local_theory* → *local_theory*
no_notation	:	*local_theory* → *local_theory*
write	:	*proof*(*state*) → *proof*(*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.

8.4 The Pure syntax

8.4.1 Lexical matters

The inner lexical syntax vaguely resembles the outer one (§3.2), 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. §8.4.2);
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 (§3.2).

$$\begin{array}{rcl} \mathit{id} & = & \mathit{short_ident} \\ \mathit{longid} & = & \mathit{long_ident} \\ \mathit{var} & = & \mathit{var} \\ \mathit{tid} & = & \mathit{type_ident} \\ \mathit{tvar} & = & \mathit{type_var} \\ \mathit{num_token} & = & \mathit{nat} \\ \mathit{float_token} & = & \mathit{nat}\texttt{.}\mathit{nat} \\ \mathit{str_token} & = & \texttt{''} \ldots \texttt{''} \\ \mathit{string_token} & = & \texttt{"} \ldots \texttt{"} \end{array}$$

$$\mathit{cartouche} \quad = \quad \texttt{\<open>} \ldots \texttt{\<close>}$$

The token categories *num_token*, *float_token*, *str_token*, *string_token*, and *cartouche* are not used in Pure. Object-logics may implement numerals and string literals by adding appropriate syntax declarations, together with some translation functions (e.g. see `~~/src/HOL/Tools/string_syntax.ML`).

The derived categories *num_const*, and *float_const*, provide robust access to the respective tokens: the syntax tree holds a syntactic constant instead of a free variable.

Formal document comments (§3.3.5) may be also used within the inner syntax.

8.4.2 Priority grammars

A context-free grammar consists of a set of *terminal symbols*, a set of *nonterminal symbols* and a set of *productions*. Productions have the form $A = \gamma$, 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 $\longrightarrow_G$ as follows. Let α and β denote strings of terminal or nonterminal symbols. Then $\alpha\ A^{(p)}\ \beta \longrightarrow_G \alpha\ \gamma\ \beta$ 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.

$$\begin{array}{rcl} A^{(1000)} & = & \texttt{(}\ A^{(0)}\ \texttt{)} \\ A^{(1000)} & = & \texttt{0} \\ A^{(0)} & = & A^{(0)}\ \texttt{+}\ A^{(1)} \\ A^{(2)} & = & A^{(3)}\ \texttt{*}\ A^{(2)} \\ A^{(3)} & = & \texttt{-}\ A^{(3)} \end{array}$$

The choice of priorities determines that `-` binds tighter than `*`, which binds tighter than `+`. Furthermore `+` associates to the left and `*` 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 (. . .) in an informal but obvious way.

Using these conventions, the example grammar specification above takes the form:

$$\begin{array}{rcll} A & = & \texttt{(}\ A\ \texttt{)} & \\ & | & \texttt{0} & \\ & | & A\ \texttt{+}\ A^{(1)} & (0) \\ & | & A^{(3)}\ \texttt{*}\ A^{(2)} & (2) \\ & | & \texttt{-}\ A^{(3)} & (3) \end{array}$$

8.4.3 The Pure grammar

The priority grammar of the *Pure* theory is defined approximately like this:

$$\begin{array}{rcll} \mathit{any} & = & \mathit{prop} \mid \mathit{logic} & \\ & & & \\ \mathit{prop} & = & \texttt{(}\ \mathit{prop}\ \texttt{)} & \\ & | & \mathit{prop}^{(4)}\ \texttt{::}\ \mathit{type} & (3) \\ & | & \mathit{any}^{(3)}\ \texttt{==}\ \mathit{any}^{(3)} & (2) \\ & | & \mathit{any}^{(3)} \equiv \mathit{any}^{(3)} & (2) \\ & | & \mathit{prop}^{(3)}\ \texttt{\&\&\&}\ \mathit{prop}^{(2)} & (2) \\ & | & \mathit{prop}^{(2)}\ \texttt{==>}\ \mathit{prop}^{(1)} & (1) \\ & | & \mathit{prop}^{(2)} \Longrightarrow \mathit{prop}^{(1)} & (1) \\ & | & \texttt{[|}\ \mathit{prop}\ \texttt{;}\ \ldots\ \texttt{;}\ \mathit{prop}\ \texttt{|]}\ \texttt{==>}\ \mathit{prop}^{(1)} & (1) \\ & | & [\![\ \mathit{prop}\ \texttt{;}\ \ldots\ \texttt{;}\ \mathit{prop}\ ]\!] \Longrightarrow \mathit{prop}^{(1)} & (1) \\ & | & \texttt{!!}\ \mathit{idts}\ \texttt{.}\ \mathit{prop} & (0) \\ & | & \bigwedge \mathit{idts}\ \texttt{.}\ \mathit{prop} & (0) \\ & | & \texttt{OFCLASS (}\ \mathit{type}\ \texttt{,}\ \mathit{logic}\ \texttt{)} & \end{array}$$

	`SORT_CONSTRAINT` (*type*)	
	`TERM` *logic*	
	`PROP` *aprop*	
aprop	= (*aprop*)	
	id | *longid* | *var* | _ | ...	
	`CONST` *id* | `CONST` *longid*	
	`XCONST` *id* | `XCONST` *longid*	
	$logic^{(1000)}$ $any^{(1000)}$ ... $any^{(1000)}$	(999)
logic	= (*logic*)	
	$logic^{(4)}$ :: *type*	(3)
	id | *longid* | *var* | _ | ...	
	`CONST` *id* | `CONST` *longid*	
	`XCONST` *id* | `XCONST` *longid*	
	$logic^{(1000)}$ $any^{(1000)}$ ... $any^{(1000)}$	(999)
	% *pttrns* . $any^{(3)}$	(3)
	λ *pttrns* . $any^{(3)}$	(3)
	(==) | (≡) | (&&&)	
	(==>) | (⟹)	
	`TYPE` (*type*)	
idt	= (*idt*) | *id* | _	
	id :: *type*	(0)
	_ :: *type*	(0)
index	= \<^bsub> $logic^{(0)}$ \<^esub> | | 1	
idts	= *idt* | $idt^{(1)}$ *idts*	(0)
pttrn	= *idt*	
pttrns	= *pttrn* | $pttrn^{(1)}$ *pttrns*	(0)
type	= (*type*)	
	tid | *tvar* | _	
	tid :: *sort* | *tvar* :: *sort* | _ :: *sort*	
	type_name | $type^{(1000)}$ *type_name*	
	(*type* , ... , *type*) *type_name*	

		$type^{(1)}$ `=>` *type*	(0)
	|	$type^{(1)} \Rightarrow type$	(0)
	|	`[` *type* `,` ... `,` *type* `]` `=>` *type*	(0)
	|	`[` *type* `,` ... `,` *type* `]` $\Rightarrow$ *type*	(0)
type_name	=	*id* | *longid*	
sort	=	*class_name* | `_` | `{}`	
	|	`{` *class_name* `,` ... `,` *class_name* `}`	
class_name	=	*id* | *longid*	

Here literal terminals are printed `verbatim`; see also §8.4.1 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 $\bigwedge$.

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 $x :: nat\ y$ is parsed as $x :: (nat\ y)$, treating y like a type constructor applied to *nat*. To avoid this interpretation, write $(x :: nat)\ y$ with explicit parentheses.
- Similarly, $x :: nat\ y :: nat$ is parsed as $x :: (nat\ y :: nat)$. The correct form is $(x :: nat)\ (y :: nat)$, or $(x :: nat)\ y :: nat$ if y is last in the sequence of identifiers.
- Type constraints for terms bind very weakly. For example, $x < y :: nat$ is normally parsed as $(x < y) :: nat$, unless $<$ has a very low priority, in which case the input is likely to be ambiguous. The correct form is $x < (y :: nat)$.
- Dummy variables (written as underscore) may occur in different roles.

  A sort "_" refers to a vacuous constraint for type variables, which is effectively ignored in type-inference.

  A type "_" or "_ :: *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 $\lambda x\ _.\ x$, which is α-equivalent to $\lambda x\ y.\ x$.

  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 §3.3.8) 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 "`...`" may be also written as ellipsis symbol `\<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 (§6.3).

`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 (§8.5.2), notably on the RHS.

`XCONST` is similar to `CONST`, but retains the constant name as given. This is only relevant to translation rules (§8.5.2), notably on the LHS.

8.4.4 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 §8.4.1.

productions lists the productions of the underlying priority grammar; see §8.4.2.

Many productions have an extra ... ⇒ *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.

print modes lists the alternative print modes provided by this grammar; see §8.1.3.

parse_rules and *print_rules* relate to syntax translations (macros); see §8.5.2.

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 §8.5.3.

parse_translation and *print_translation* list the sets of constants that invoke regular translation functions; see §8.5.3.

8.4.5 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.

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.

8.5 Syntax transformations

The inner syntax engine of Isabelle provides separate mechanisms to transform parse trees either via rewrite systems on first-order ASTs (§8.5.2), or ML functions on ASTs or syntactic λ-terms (§8.5.3). This works both for parsing and printing, as outlined in figure 8.1.

These intermediate syntax tree formats eventually lead to a pre-term with all names and binding scopes resolved, but most type information still missing. Explicit type constraints might be given by the user, or implicit position

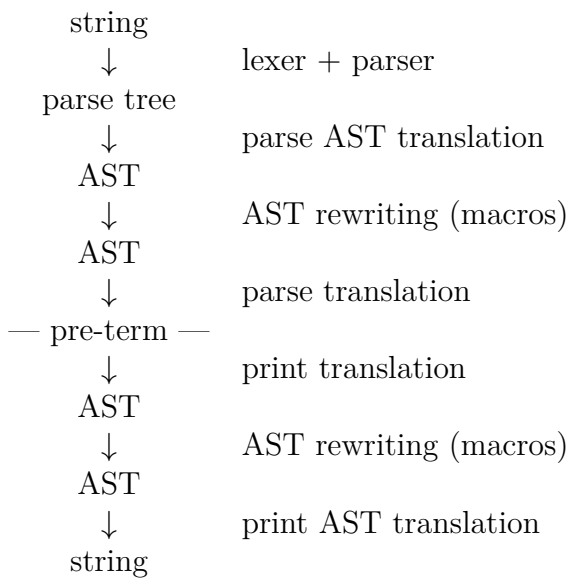


Figure 8.1: Parsing and printing with translations

information by the system — both need to be passed-through carefully by syntax transformations.

Pre-terms are further processed by the so-called *check* and *uncheck* phases that are intertwined with type-inference (see also [55]). The latter allows to operate on higher-order abstract syntax with proper binding and type information already available.

As a rule of thumb, anything that manipulates bindings of variables or constants needs to be implemented as syntax transformation (see below). Anything else is better done via check/uncheck: a prominent example application is the **abbreviation** concept of Isabelle/Pure.

8.5.1 Abstract syntax trees

The ML datatype `Ast.ast` explicitly represents the intermediate AST format that is used for syntax rewriting (§8.5.2). It is defined in ML as follows:

```
datatype ast =
  Constant of string |
  Variable of string |
  Appl of ast list
```

An AST is either an atom (constant or variable) or a list of (at least two) subtrees. Occasional diagnostic output of ASTs uses notation that resembles

S-expression of LISP. Constant atoms are shown as quoted strings, variable atoms as non-quoted strings and applications as a parenthesized list of subtrees. For example, the AST

```
Ast.Appl [Ast.Constant "_abs", Ast.Variable "x", Ast.Variable "t"]
```

is pretty-printed as `("_abs" x t)`. Note that `()` and `(x)` are excluded as ASTs, because they have too few subtrees.

AST application is merely a pro-forma mechanism to indicate certain syntactic structures. Thus `(c a b)` could mean either term application or type application, depending on the syntactic context.

Nested application like `(("_abs" x t) u)` is also possible, but ASTs are definitely first-order: the syntax constant `"_abs"` does not bind the `x` in any way. Proper bindings are introduced in later stages of the term syntax, where `("_abs" x t)` becomes an `Abs` node and occurrences of `x` in `t` are replaced by bound variables (represented as de-Bruijn indices).

AST constants versus variables

Depending on the situation — input syntax, output syntax, translation patterns — the distinction of atomic ASTs as `Ast.Constant` versus `Ast.Variable` serves slightly different purposes.

Input syntax of a term such as $f\ a\ b = c$ does not yet indicate the scopes of atomic entities f, a, b, c: they could be global constants or local variables, even bound ones depending on the context of the term. `Ast.Variable` leaves this choice still open: later syntax layers (or translation functions) may capture such a variable to determine its role specifically, to make it a constant, bound variable, free variable etc. In contrast, syntax translations that introduce already known constants would rather do it via `Ast.Constant` to prevent accidental re-interpretation later on.

Output syntax turns term constants into `Ast.Constant` and variables (free or schematic) into `Ast.Variable`. This information is precise when printing fully formal λ-terms.

AST translation patterns (§8.5.2) that represent terms cannot distinguish constants and variables syntactically. Explicit indication of *CONST* c inside the term language is required, unless c is known as special *syntax constant* (see also **syntax**). It is also possible to use **syntax** declarations (without mixfix annotation) to enforce that certain unqualified names are always treated as constant within the syntax machinery.

The situation is simpler for ASTs that represent types or sorts, since the concrete syntax already distinguishes type variables from type constants (constructors). So (*'a*, *'b*) *foo* corresponds to an AST application of some constant for *foo* and variable arguments for *'a* and *'b*. Note that the postfix application is merely a feature of the concrete syntax, while in the AST the constructor occurs in head position.

Authentic syntax names

Naming constant entities within ASTs is another delicate issue. Unqualified names are resolved in the name space tables in the last stage of parsing, after all translations have been applied. Since syntax transformations do not know about this later name resolution, there can be surprises in boundary cases.

Authentic syntax names for `Ast.Constant` avoid this problem: the fully-qualified constant name with a special prefix for its formal category (*class*, *type*, *const*, *fixed*) represents the information faithfully within the untyped AST format. Accidental overlap with free or bound variables is excluded as well. Authentic syntax names work implicitly in the following situations:

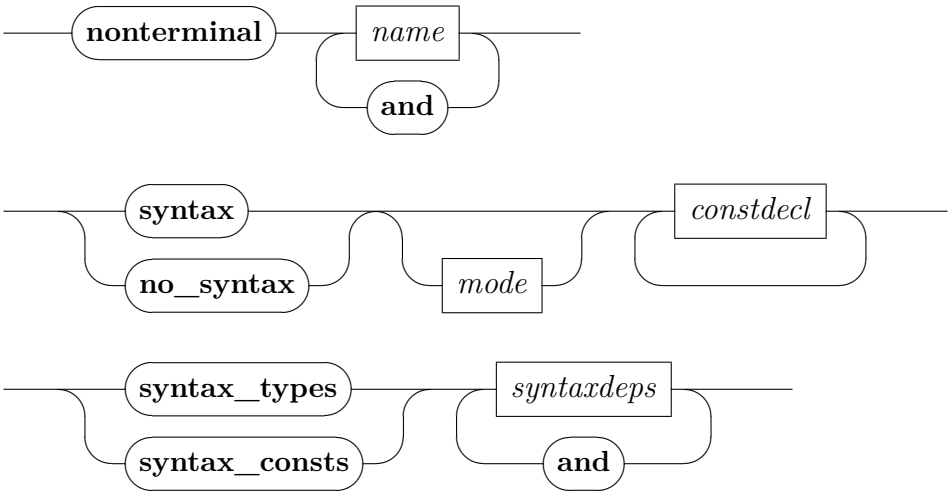
- Input of term constants (or fixed variables) that are introduced by concrete syntax via **notation**: the correspondence of a particular grammar production to some known term entity is preserved.
- Input of type constants (constructors) and type classes — thanks to explicit syntactic distinction independently on the context.
- Output of term constants, type constants, type classes — this information is already available from the internal term to be printed.

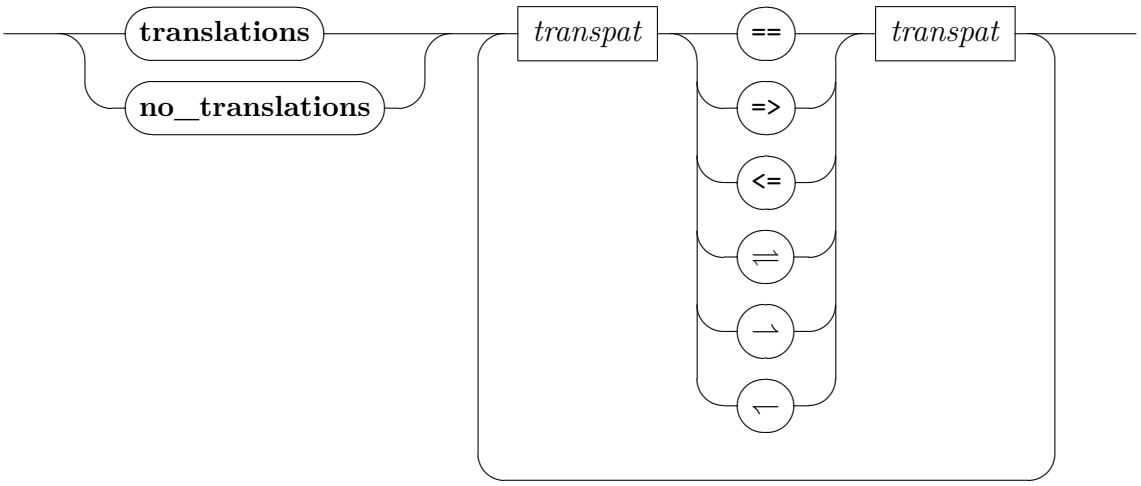
In other words, syntax transformations that operate on input terms written as prefix applications are difficult to make robust. Luckily, this case rarely occurs in practice, because syntax forms to be translated usually correspond to some concrete notation.

8.5.2 Raw syntax and translations

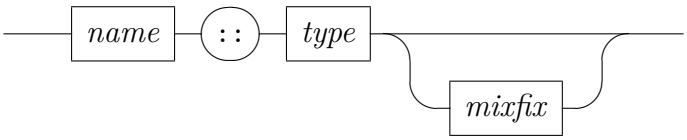
nonterminal	:	*theory* → *theory*	
syntax	:	*local_theory* → *local_theory*	
no_syntax	:	*local_theory* → *local_theory*	
syntax_types	:	*local_theory* → *local_theory*	
syntax_consts	:	*local_theory* → *local_theory*	
translations	:	*local_theory* → *local_theory*	
no_translations	:	*local_theory* → *local_theory*	
syntax_ast_trace	:	*attribute*	default *false*
syntax_ast_stats	:	*attribute*	default *false*

Unlike mixfix notation for existing formal entities (§8.3), raw syntax declarations provide full access to the priority grammar of the inner syntax, without any sanity checks. This includes additional syntactic categories (via **nonterminal**) and free-form grammar productions (via **syntax** with formal dependencies via **syntax_types** and **syntax_consts**). Additional syntax translations (or macros, via **translations**) are required to turn resulting parse trees into proper representations of formal entities again.

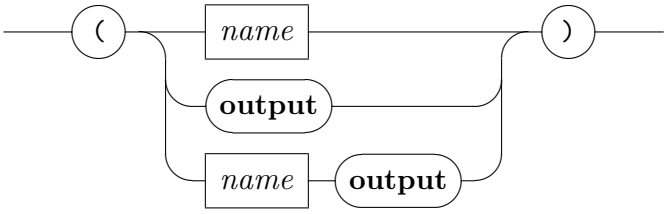




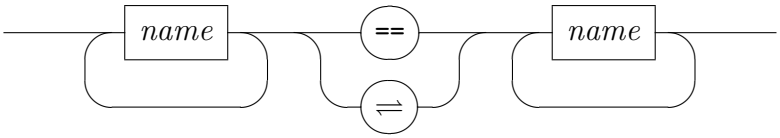
constdecl



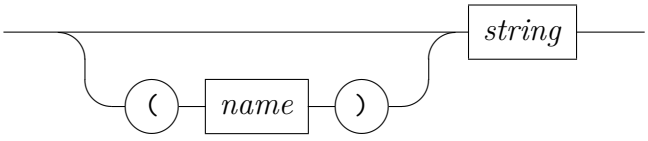
mode



syntaxdeps



transpat



nonterminal *c* declares a type constructor *c* (without arguments) to act as purely syntactic type: a nonterminal symbol of the inner syntax.

syntax (*mode*) $c :: \sigma$ (*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 §8.2, the mixfix annotation *mx* = *template ps q* together with type $\sigma = \tau_1 \Rightarrow \ldots \tau_n \Rightarrow \tau$ 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 $\tau_i = prop$
- *logic* if $\tau_i = (\ldots)\kappa$ for logical type constructor $\kappa \neq prop$
- *any* if $\tau_i = \alpha$ for type variables
- κ if $\tau_i = \kappa$ for nonterminal κ (syntactic type constructor)

Each A_i is decorated by priority p_i from the given list *ps*; missing 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. *_foobar*). Names should be chosen with care, to avoid clashes with other 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.

syntax_types *syntax* $\rightleftharpoons$ *types* and **syntax_consts** *syntax* $\rightleftharpoons$ *consts* declare dependencies of syntax constants wrt. formal entities of the logic: multiple names may be given on both sides. This tells the inner-syntax engine how to markup concrete syntax, to support hyperlinks in the browser or editor. It is essentially an abstract specification of the effect of translation rules (see below) or translation functions (see §8.5.3).

translations *rules* specifies syntactic translation rules (i.e. macros) as first-order rewrite rules on ASTs (§8.5.1). The theory context maintains two independent lists translation rules: parse rules (`=>` or $\rightharpoonup$) and print rules (`<=` or $\leftharpoondown$). For convenience, both can be specified simultaneously as parse / print rules (`==` or $\rightleftharpoons$).

Translation patterns may be prefixed by the syntactic category to be used for parsing; the default is *logic* which means that regular term syntax is used. Both sides of the syntax translation rule undergo parsing and parse AST translations §8.5.3, in order to perform some fundamental normalization like $\lambda x\ y.\ b \rightsquigarrow \lambda x.\ \lambda y.\ b$, but other AST translation rules are *not* applied recursively here.

When processing AST patterns, the inner syntax lexer runs in a different mode that allows identifiers to start with underscore. This accommodates the usual naming convention for auxiliary syntax constants — those that do not have a logical counter part — by allowing to specify arbitrary AST applications within the term syntax, independently of the corresponding concrete syntax.

Atomic ASTs are distinguished as `Ast.Constant` versus `Ast.Variable` as follows: a qualified name or syntax constant declared via **syntax**, or parse tree head of concrete notation becomes `Ast.Constant`, anything else `Ast.Variable`. Note that *CONST* and *XCONST* within the term language (§8.4.3) allow to enforce treatment as constants.

AST rewrite rules (*lhs*, *rhs*) need to obey the following side-conditions:

- Rules must be left linear: *lhs* must not contain repeated variables.[2]
- Every variable in *rhs* must also occur in *lhs*.

no_translations *rules* removes syntactic translation rules, which are interpreted in the same manner as for **translations** above.

syntax_ast_trace and *syntax_ast_stats* control diagnostic output in the AST normalization process, when translation rules are applied to concrete input or output.

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

[2]The deeper reason for this is that AST equality is not well-defined: different occurrences of the "same" AST could be decorated differently by accidental type-constraints or source position information, for example.

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 *HOL.List*.
- Change of binding status of variables: anything beyond the built-in **binder** mixfix annotation requires explicit syntax translations. For example, consider the set comprehension syntax $\{x.\ P\}$ as defined in theory *HOL.Set*.

Applying translation rules

As a term is being parsed or printed, an AST is generated as an intermediate form according to figure 8.1. The AST is normalized by applying translation rules in the manner of a first-order term rewriting system. We first examine how a single rule is applied.

Let t be the abstract syntax tree to be normalized and (*lhs*, *rhs*) some translation rule. A subtree u of t is called *redex* if it is an instance of *lhs*; in this case the pattern *lhs* is said to match the object u. A redex matched by *lhs* may be replaced by the corresponding instance of *rhs*, thus *rewriting* the AST t. Matching requires some notion of *place-holders* in rule patterns: `Ast.Variable` serves this purpose.

More precisely, the matching of the object u against the pattern *lhs* is performed as follows:

- Objects of the form `Ast.Variable` x or `Ast.Constant` x are matched by pattern `Ast.Constant` x. Thus all atomic ASTs in the object are treated as (potential) constants, and a successful match makes them actual constants even before name space resolution (see also §8.5.1).
- Objects of the form `Ast.Appl [Constant "_constrain",` u`,` T`]`, for u as `Ast.Variable` x or `Ast.Constant` x, are matched by `Ast.Constant` x if the AST T encodes a source position (from parsing) or if types are considered optional (for printing).
- Object u is matched by pattern `Ast.Variable` x, binding x to u.
- Object `Ast.Appl` *us* is matched by `Ast.Appl` *ts* if *us* and *ts* have the same length and each corresponding subtree matches.

- In every other case, matching fails.

A successful match yields a substitution that is applied to *rhs*, generating the instance that replaces *u*.

Normalizing an AST involves repeatedly applying translation rules until none are applicable. This works yoyo-like: top-down, bottom-up, top-down, etc. At each subtree position, rules are chosen in order of appearance in the theory definitions.

The configuration options *syntax_ast_trace* and *syntax_ast_stats* might help to understand this process and diagnose problems.

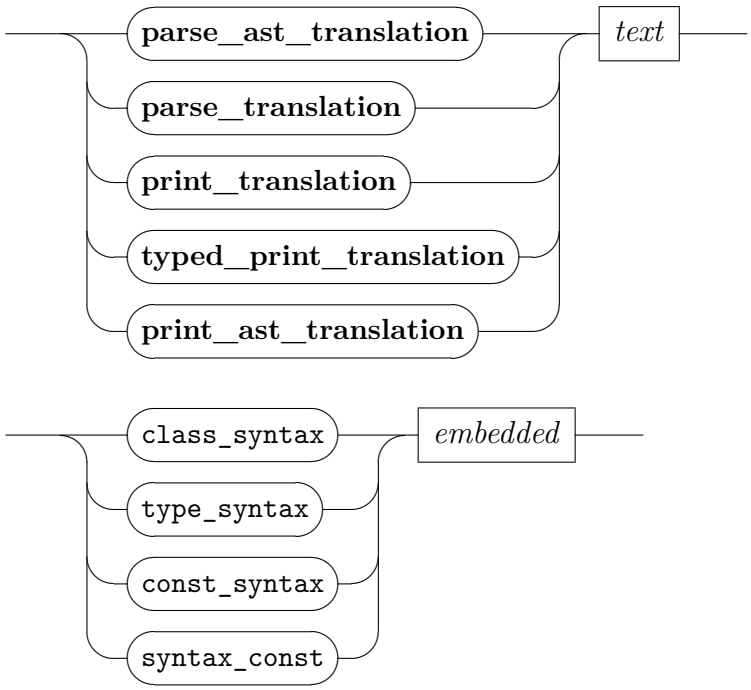
! If syntax translation rules work incorrectly, the output of **print_syntax** with its *rules* sections reveals the actual internal forms of AST pattern, without potentially confusing concrete syntax. Recall that AST constants appear as quoted strings and variables without quotes.

! If *eta_contract* is set to *true*, terms will be η-contracted *before* the AST rewriter sees them. Thus some abstraction nodes needed for print rules to match may vanish. For example, *Ball A* $(\lambda x.\ P\ x)$ would contract to *Ball A P* and the standard print rule would fail to apply. This problem can be avoided by hand-written ML translation functions (see also §8.5.3), which is in fact the same mechanism used in built-in **binder** declarations.

8.5.3 Syntax translation functions

parse_ast_translation	:	*theory* → *theory*
parse_translation	:	*theory* → *theory*
print_translation	:	*theory* → *theory*
typed_print_translation	:	*theory* → *theory*
print_ast_translation	:	*theory* → *theory*
class_syntax	:	*ML antiquotation*
type_syntax	:	*ML antiquotation*
const_syntax	:	*ML antiquotation*
syntax_const	:	*ML antiquotation*

Syntax translation functions written in ML admit almost arbitrary manipulations of inner syntax, at the expense of some complexity and obscurity in the implementation.



parse_translation etc. declare syntax translation functions to the theory. Any of these commands have a single *text* argument that refers to an ML expression of appropriate type as follows:

```
parse_ast_translation :
  (string * (Proof.context -> Ast.ast list -> Ast.ast)) list
parse_translation :
  (string * (Proof.context -> term list -> term)) list
print_translation :
  (string * (Proof.context -> term list -> term)) list
typed_print_translation :
  (string * (Proof.context -> typ -> term list -> term)) list
print_ast_translation :
  (string * (Proof.context -> Ast.ast list -> Ast.ast)) list
```

The argument list consists of (c, tr) pairs, where c is the syntax name of the formal entity involved, and tr a function that translates a syntax form $c\ args$ into $tr\ ctxt\ args$ (depending on the context). The Isabelle/ML naming convention for parse translations is c_tr and for print translations c_tr'.

The **print_syntax** command displays the sets of names associated with the translation functions of a theory under *parse_ast_translation* etc.

@{*class_syntax c*}, @{*type_syntax c*}, @{*const_syntax c*} inline the authentic syntax name of the given formal entities into the ML source. This is the fully-qualified logical name prefixed by a special marker to indicate its kind: thus different logical name spaces are properly distinguished within parse trees.

@{*const_syntax c*} inlines the name c of the given syntax constant, having checked that it has been declared via some **syntax** commands within the theory context. Note that the usual naming convention makes syntax constants start with underscore, to reduce the chance of accidental clashes with other names occurring in parse trees (unqualified constants etc.).

The translation strategy

The different kinds of translation functions are invoked during the transformations between parse trees, ASTs and syntactic terms (cf. figure 8.1). Whenever a combination of the form $c\ x_1\ \ldots\ x_n$ is encountered, and a translation function f of appropriate kind is declared for c, the result is produced by evaluation of $f\ [x_1, \ldots, x_n]$ in ML.

For AST translations, the arguments $x_1, \ldots, x_n$ are ASTs. A combination has the form `Ast.Constant` c or `Ast.Appl` [`Ast.Constant` c, $x_1, \ldots, x_n$]. For term translations, the arguments are terms and a combination has the form `Const` (c, τ) or `Const` (c, τ) \$ x_1 \$ ... \$ x_n. Terms allow more sophisticated transformations than ASTs do, typically involving abstractions and bound variables. *Typed* print translations may even peek at the type τ of the constant they are invoked on, although some information might have been suppressed for term output already.

Regardless of whether they act on ASTs or terms, translation functions called during the parsing process differ from those for printing in their overall behaviour:

Parse translations are applied bottom-up. The arguments are already in translated form. The translations must not fail; exceptions trigger an error message. There may be at most one function associated with any syntactic name.

Print translations are applied top-down. They are supplied with arguments that are partly still in internal form. The result again undergoes translation; therefore a print translation should not introduce as head

the very constant that invoked it. The function may raise exception `Match` to indicate failure; in this event it has no effect. Multiple functions associated with some syntactic name are tried in the order of declaration in the theory.

Only constant atoms — constructor `Ast.Constant` for ASTs and `Const` for terms — can invoke translation functions. This means that parse translations can only be associated with parse tree heads of concrete syntax, or syntactic constants introduced via other translations. For plain identifiers within the term language, the status of constant versus variable is not yet know during parsing. This is in contrast to print translations, where constants are explicitly known from the given term in its fully internal form.

8.5.4 Built-in syntax transformations

Here are some further details of the main syntax transformation phases of figure 8.1.

Transforming parse trees to ASTs

The parse tree is the raw output of the parser. It is transformed into an AST according to some basic scheme that may be augmented by AST translation functions as explained in §8.5.3.

The parse tree is constructed by nesting the right-hand sides of the productions used to recognize the input. Such parse trees are simply lists of tokens and constituent parse trees, the latter representing the nonterminals of the productions. Ignoring AST translation functions, parse trees are transformed to ASTs by stripping out delimiters and copy productions, while retaining some source position information from input tokens.

The Pure syntax provides predefined AST translations to make the basic λ-term structure more apparent within the (first-order) AST representation, and thus facilitate the use of **translations** (see also §8.5.2). This covers ordinary term application, type application, nested abstraction, iterated meta implications and function types. The effect is illustrated on some representative input strings is as follows:

input source	AST
$f\ x\ y\ z$	`(f x y z)`
$'a\ ty$	`(ty 'a)`
$('a, 'b)ty$	`(ty 'a 'b)`
$\lambda x\ y\ z.\ t$	`("_abs" x ("_abs" y ("_abs" z t)))`
$\lambda x :: 'a.\ t$	`("_abs" ("_constrain" x 'a) t)`
$[\![P; Q; R]\!] \Longrightarrow S$	`("Pure.imp" P ("Pure.imp" Q ("Pure.imp" R S)))`
$['a, 'b, 'c] \Rightarrow 'd$	`("fun" 'a ("fun" 'b ("fun" 'c 'd)))`

Note that type and sort constraints may occur in many other places — translations need to cope with them. The built-in syntax transformation from parse trees to ASTs insert additional constraints that represent source positions.

Transforming ASTs to terms

After application of macros (§8.5.2), the AST is transformed into a term. This term still lacks proper type information, but it might contain some constraints consisting of applications with head `_constrain`, where the second argument is a type encoded as a pre-term within the syntax. Type inference later introduces correct types, or indicates type errors in the input.

Ignoring parse translations, ASTs are transformed to terms by mapping AST constants to term constants, AST variables to term variables or constants (according to the name space), and AST applications to iterated term applications.

The outcome is still a first-order term. Proper abstractions and bound variables are introduced by parse translations associated with certain syntax constants. Thus `("_abs" x x)` eventually becomes a de-Bruijn term `Abs ("x", _, Bound 0)`.

Printing of terms

The output phase is essentially the inverse of the input phase. Terms are translated via abstract syntax trees into pretty-printed text.

Ignoring print translations, the transformation maps term constants, variables and applications to the corresponding constructs on ASTs. Abstractions are mapped to applications of the special constant `_abs` as seen before. Type constraints are represented via special `_constrain` forms, according to various policies of type annotation determined elsewhere. Sort constraints of type variables are handled in a similar fashion.

After application of macros (§8.5.2), the AST is finally pretty-printed. The built-in print AST translations reverse the corresponding parse AST translations.

For the actual printing process, the priority grammar (§8.4.2) plays a vital role: productions are used as templates for pretty printing, with argument slots stemming from nonterminals, and syntactic sugar stemming from literal tokens.

Each AST application with constant head c and arguments t_1, ..., t_n (for $n = 0$ the AST is just the constant c itself) is printed according to the first grammar production of result name c. The required syntax priority of the argument slot is given by its nonterminal $A^{(p)}$. The argument t_i that corresponds to the position of $A^{(p)}$ is printed recursively, and then put in parentheses *if* its priority p requires this. The resulting output is concatenated with the syntactic sugar according to the grammar production.

If an AST application $(c\ x_1\ \ldots\ x_m)$ has more arguments than the corresponding production, it is first split into $((c\ x_1\ \ldots\ x_n)\ x_{n+1}\ \ldots\ x_m)$ and then printed recursively as above.

Applications with too few arguments or with non-constant head or without a corresponding production are printed in prefix-form like $f\ t_1\ \ldots\ t_n$ for terms.

Multiple productions associated with some name c are tried in order of appearance within the grammar. An occurrence of some AST variable x is printed as x outright.

White space is *not* inserted automatically. If blanks (or breaks) are required to separate tokens, they need to be specified in the mixfix declaration (§8.2).

Chapter 9

Generic tools and packages

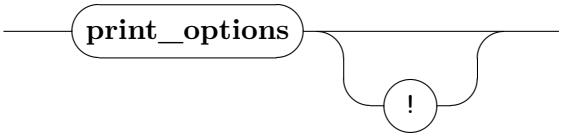
9.1 Configuration options

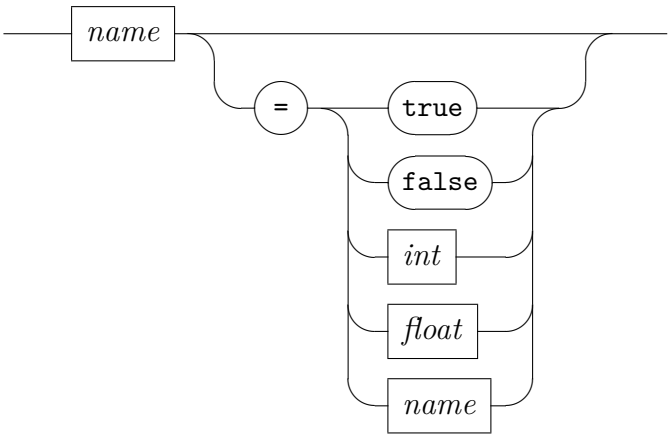
Isabelle/Pure maintains a record of named configuration options within the theory or proof context, with values of type `bool`, `int`, `real`, or `string`. Tools may declare options in ML, and then refer to these values (relative to the context). Thus global reference variables are easily avoided. The 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 **declare** or **using** like this:

declare [[*show_main_goal* = *false*]]

notepad
begin
  note [[*show_main_goal* = *true*]]
end

print_options : *context* →





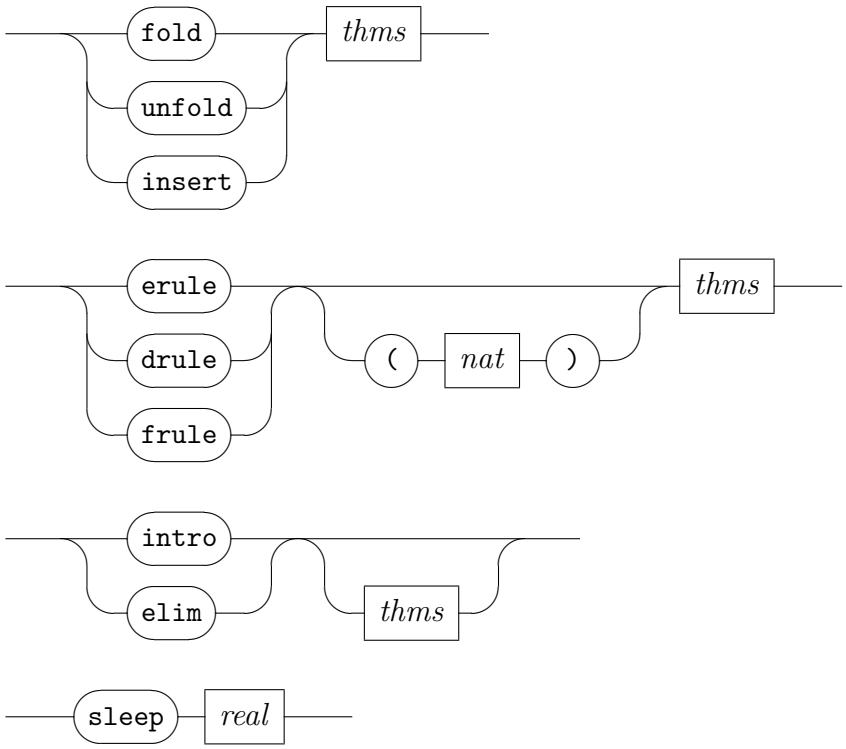
print_options prints the available configuration options, with names, types, and current values; the "!" option indicates extra verbosity.

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** : *method*
*drule** : *method*
*frule** : *method*
intro : *method*
elim : *method*
fail : *method*
succeed : *method*
sleep : *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.

Unfolding works in two stages: first, the given equations are used directly for rewriting; second, the equations are passed through the attribute *abs_def* before rewriting — to ensure that definitions are fully expanded, regardless of the actual parameters that are provided.

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 §6.4.3), but apply rules by elim-resolution, destruct-resolution, and forward-resolution, respectively [55]. 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.

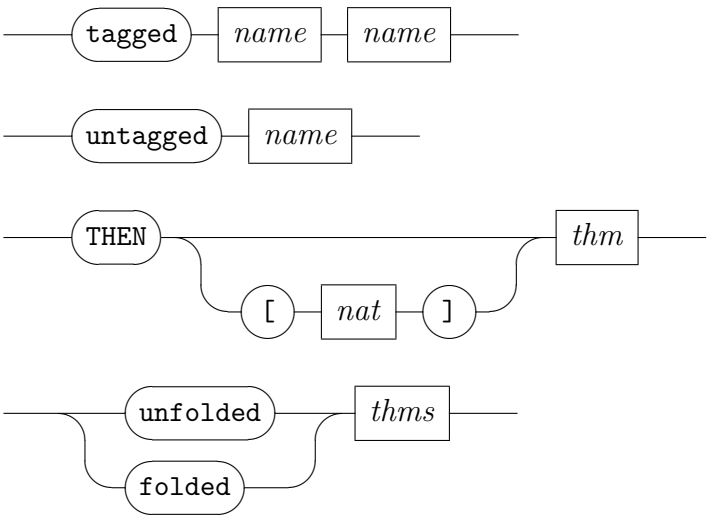
intro and *elim* repeatedly refine some goal by intro- or elim-resolution, after having inserted any chained facts. Exactly the rules given as arguments are taken into account; this allows fine-tuned decomposition of a proof problem, in contrast to common automated tools.

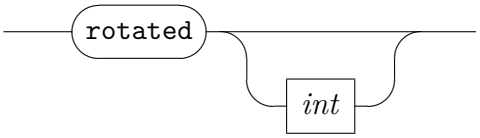
fail yields an empty result sequence; it is the identity of the "|" method combinator (cf. §6.4.1).

succeed yields a single (unchanged) result; it is the identity of the "," method combinator (cf. §6.4.1).

sleep s succeeds after a real-time delay of *s* seconds. This is occasionally useful for demonstration and testing purposes.

tagged : *attribute*
untagged : *attribute*
THEN : *attribute*
unfolded : *attribute*
folded : *attribute*
abs_def : *attribute*
rotated : *attribute*
elim_format : *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 composes rules by resolution; it resolves with the first premise of a (an alternative position may be also specified). See also `RS` in [55].

unfolded $a_1 \ldots a_n$ and *folded* $a_1 \ldots a_n$ 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* steps always expand it. This also works for object-logic equality.

rotated n rotate the premises of a theorem by n (default 1).

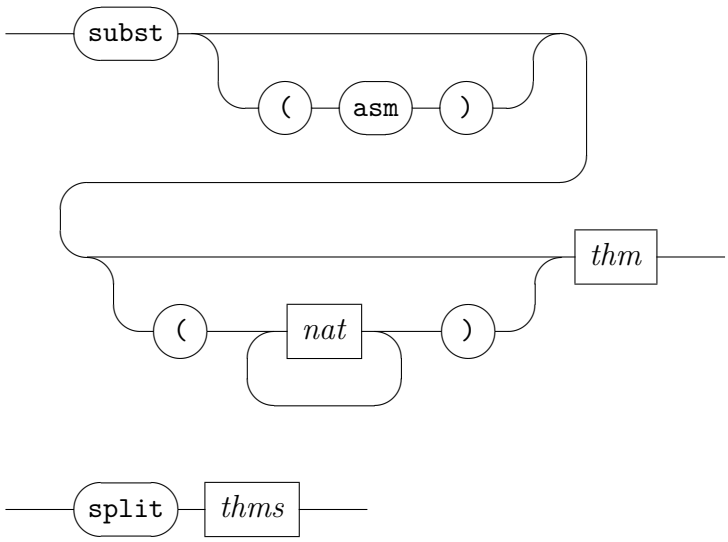
elim_format turns a destruction rule into elimination rule format, by resolving with the rule $PROP\ A \Longrightarrow (PROP\ A \Longrightarrow PROP\ B) \Longrightarrow PROP\ B$.

Note that the Classical Reasoner (§9.4) provides its own version of this operation.

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 §6.3), 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.

split $a_1 \ldots a_n$ 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 *simp* method already involves repeated application of split rules as declared in the current context, using *split*, for example.

9.3 The Simplifier

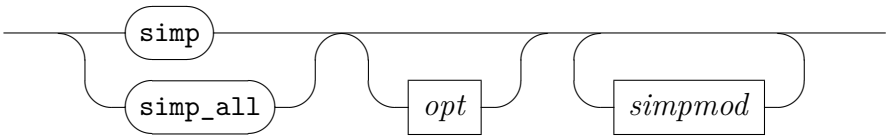
The Simplifier performs conditional and unconditional rewriting and uses contextual information: rule declarations in the background theory or local proof context are taken into account, as well as chained facts and subgoal premises ("local assumptions"). There are several general hooks that allow to modify the simplification strategy, or incorporate other proof tools that solve sub-problems, produce rewrite rules on demand etc.

The rewriting strategy is always strictly bottom up, except for congruence rules, which are applied while descending into a term. Conditions in conditional rewrite rules are solved recursively before the rewrite rule is applied.

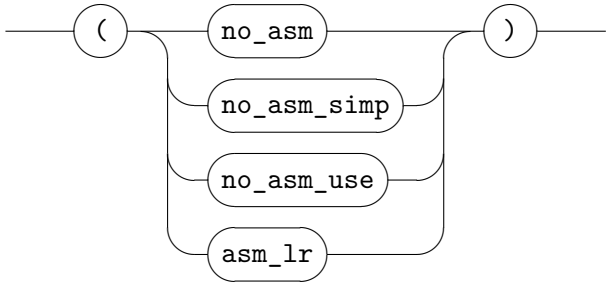
The default Simplifier setup of major object logics (HOL, HOLCF, FOL, ZF) makes the Simplifier ready for immediate use, without engaging into the internal structures. Thus it serves as general-purpose proof tool with the main focus on equational reasoning, and a bit more than that.

9.3.1 Simplification methods

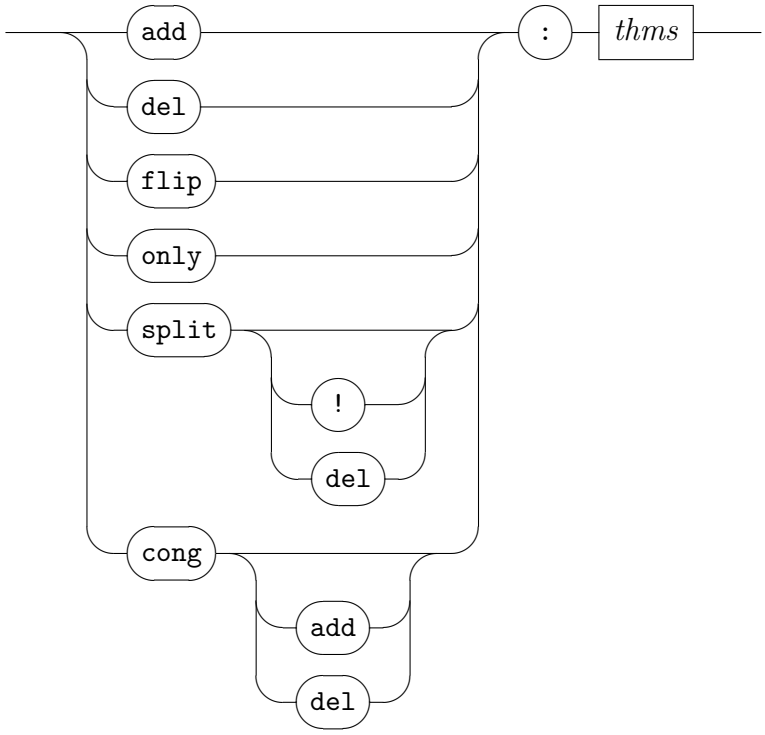
simp	:	*method*	
simp_all	:	*method*	
Pure.simp	:	*method*	
Pure.simp_all	:	*method*	
simp_depth_limit	:	*attribute*	default 100



opt



simpmod



simp invokes the Simplifier on the first subgoal, after inserting chained facts as additional goal premises; further rule declarations may be included via (*simp add*: *facts*). The proof method fails if the subgoal remains unchanged after simplification.

Note that the original goal premises and chained facts are subject to simplification themselves, while declarations via *add*/*del* merely follow the policies of the object-logic to extract rewrite rules from theorems, without further simplification. This may lead to slightly different behavior in either case, which might be required precisely like that in some boundary situations to perform the intended simplification step!

Modifier *flip* deletes the following theorems from the simpset and adds their symmetric version (i.e. lhs and rhs exchanged). No warning is shown if the original theorem was not present.

The *only* modifier first removes all other rewrite rules, looper tactics (including split rules), congruence rules, and then behaves like *add*. Implicit solvers remain, which means that trivial rules like reflexivity or introduction of *True* are available to solve the simplified subgoals, but also non-trivial tools like linear arithmetic in HOL. The latter may lead to some surprise of the meaning of "only" in Isabelle/HOL compared to English!

The *split* modifiers add or delete rules for the Splitter (see also §9.3.6 on the looper). 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). The ! option causes the split rules to be used aggressively: after each application of a split rule in the conclusion, the *safe* tactic of the classical reasoner (see §9.4.5) is applied to the new goal. The net effect is that the goal is split into the different cases. This option can speed up simplification of goals with many nested conditional or case expressions significantly.

There is also a separate *split* method available for single-step case splitting. The effect of repeatedly applying (*split thms*) can be imitated by "(*simp only*: *split*: *thms*)".

The *cong* modifiers add or delete Simplifier congruence rules (see also §9.3.2); the default is to add.

simp_all is similar to *simp*, but acts on all goals, working backwards from the last to the first one as usual in Isabelle.[1]

Chained facts are inserted into all subgoals, before the simplification process starts. Further rule declarations are the same as for *simp*.

The proof method fails if all subgoals remain unchanged after simplification.

simp_depth_limit limits the number of recursive invocations of the Simplifier during conditional rewriting.

[1]The order is irrelevant for goals without schematic variables, so simplification might actually be performed in parallel here.

By default the Simplifier methods above take local assumptions fully into account, using equational assumptions in the subsequent normalization process, or simplifying assumptions themselves. Further options allow to fine-tune the behavior of the Simplifier in this respect, corresponding to a variety of ML tactics as follows.[2]

Isar method	ML tactic	behavior
(*simp* (*no_asm*))	`simp_tac`	assumptions are ignored completely
(*simp* (*no_asm_simp*))	`asm_simp_tac`	assumptions are used in the simplification of the conclusion but are not themselves simplified
(*simp* (*no_asm_use*))	`full_simp_tac`	assumptions are simplified but are not used in the simplification of each other or the conclusion
(*simp*)	`asm_full_simp_tac`	assumptions are used in the simplification of the conclusion and to simplify other assumptions
(*simp* (*asm_lr*))	`asm_lr_simp_tac`	compatibility mode: an assumption is only used for simplifying assumptions which are to the right of it

In Isabelle/Pure, proof methods *simp* and *simp_all* only know about meta-equality $\equiv$. Any new object-logic needs to re-define these methods via `Simplifier.method_setup` in ML: Isabelle/FOL or Isabelle/HOL may serve as blue-prints.

Examples

We consider basic algebraic simplifications in Isabelle/HOL. The rather trivial goal $0 + (x + 0) = x + 0 + 0$ looks like a good candidate to be solved by a single call of *simp*:

lemma $0 + (x + 0) = x + 0 + 0$ **apply** *simp?* **oops**

[2]Unlike the corresponding Isar proof methods, the ML tactics do not insist in changing the goal state.

The above attempt *fails*, because 0 and (+) in the HOL library are declared as generic type class operations, without stating any algebraic laws yet. More specific types are required to get access to certain standard simplifications of the theory context, e.g. like this:

lemma fixes $x :: nat$ **shows** $0 + (x + 0) = x + 0 + 0$ **by** *simp*
lemma fixes $x :: int$ **shows** $0 + (x + 0) = x + 0 + 0$ **by** *simp*
lemma fixes $x :: 'a :: monoid_add$ **shows** $0 + (x + 0) = x + 0 + 0$ **by** *simp*

In many cases, assumptions of a subgoal are also needed in the simplification process. For example:

lemma fixes $x :: nat$ **shows** $x = 0 \Longrightarrow x + x = 0$ **by** *simp*
lemma fixes $x :: nat$ **assumes** $x = 0$ **shows** $x + x = 0$ **apply** *simp* **oops**
lemma fixes $x :: nat$ **assumes** $x = 0$ **shows** $x + x = 0$ **using** *assms* **by** *simp*

As seen above, local assumptions that shall contribute to simplification need to be part of the subgoal already, or indicated explicitly for use by the subsequent method invocation. Both too little or too much information can make simplification fail, for different reasons.

In the next example the malicious assumption $\bigwedge x{::}nat.\ f\ x = g\ (f\ (g\ x))$ does not contribute to solve the problem, but makes the default *simp* method loop: the rewrite rule $f\ ?x \equiv g\ (f\ (g\ ?x))$ extracted from the assumption does not terminate. The Simplifier notices certain simple forms of nontermination, but not this one. The problem can be solved nonetheless, by ignoring assumptions via special options as explained before:

lemma $(\bigwedge x{::}nat.\ f\ x = g\ (f\ (g\ x))) \Longrightarrow f\ 0 = f\ 0 + 0$
  by (*simp* (*no_asm*))

The latter form is typical for long unstructured proof scripts, where the control over the goal content is limited. In structured proofs it is usually better to avoid pushing too many facts into the goal state in the first place. Assumptions in the Isar proof context do not intrude the reasoning if not used explicitly. This is illustrated for a toplevel statement and a local proof body as follows:

lemma
  assumes $\bigwedge x{::}nat.\ f\ x = g\ (f\ (g\ x))$
  shows $f\ 0 = f\ 0 + 0$ **by** *simp*

notepad
begin
  assume $\bigwedge x{::}nat.\ f\ x = g\ (f\ (g\ x))$
  have $f\ 0 = f\ 0 + 0$ **by** *simp*

end

Because assumptions may simplify each other, there can be very subtle cases of nontermination. For example, the regular *simp* method applied to $P\ (f\ x) \Longrightarrow y = x \Longrightarrow f\ x = f\ y \Longrightarrow Q$ gives rise to the infinite reduction sequence

$$P\ (f\ x) \overset{f\ x \equiv f\ y}{\longmapsto} P\ (f\ y) \overset{y \equiv x}{\longmapsto} P\ (f\ x) \overset{f\ x \equiv f\ y}{\longmapsto} \ldots$$

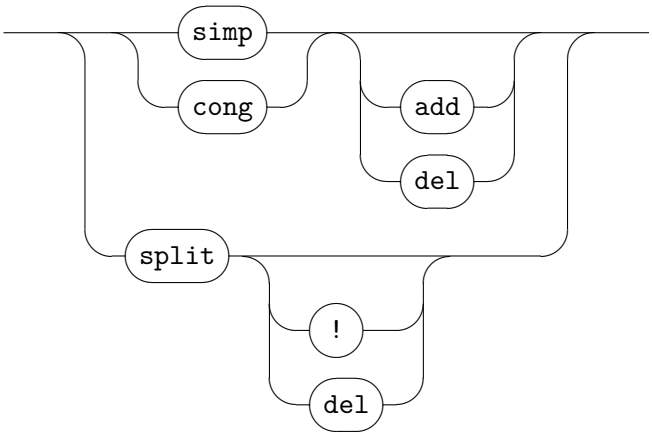
whereas applying the same to $y = x \Longrightarrow f\ x = f\ y \Longrightarrow P\ (f\ x) \Longrightarrow Q$ terminates (without solving the goal):

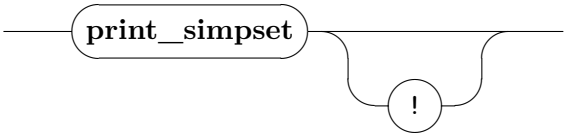
lemma $y = x \Longrightarrow f\ x = f\ y \Longrightarrow P\ (f\ x) \Longrightarrow Q$
  apply *simp*
  oops

See also §9.3.4 for options to enable Simplifier trace mode, which often helps to diagnose problems with rewrite systems.

9.3.2 Declaring rules

simp : *attribute*
split : *attribute*
cong : *attribute*
print_simpset* : *context* $\rightarrow$





simp declares rewrite rules, by adding or deleting them from the simpset within the theory or proof context. Rewrite rules are theorems expressing some form of equality, for example:

$Suc\ ?m + ?n = ?m + Suc\ ?n$
$?P \wedge ?P \longleftrightarrow ?P$
$?A \cup ?B \equiv \{x.\ x \in ?A \vee x \in ?B\}$

Conditional rewrites such as $?m < ?n \Longrightarrow ?m\ div\ ?n = 0$ are also permitted; the conditions can be arbitrary formulas.

Internally, all rewrite rules are translated into Pure equalities, theorems with conclusion *lhs* $\equiv$ *rhs*. The simpset contains a function for extracting equalities from arbitrary theorems, which is usually installed when the object-logic is configured initially. For example, $\neg\ ?x \in \{\}$ could be turned into $?x \in \{\} \equiv False$. Theorems that are declared as *simp* and local assumptions within a goal are treated uniformly in this respect.

The Simplifier accepts the following formats for the *lhs* term:

1. First-order patterns, considering the sublanguage of application of constant operators to variable operands, without λ-abstractions or functional variables. For example:
   $(?x + ?y) + ?z \equiv ?x + (?y + ?z)$
   $f\ (f\ ?x\ ?y)\ ?z \equiv f\ ?x\ (f\ ?y\ ?z)$
2. Higher-order patterns in the sense of [36]. These are terms in β-normal form (this will always be the case unless you have done something strange) where each occurrence of an unknown is of the form $?F\ x_1\ \ldots\ x_n$, where the x_i are distinct bound variables.
   For example, $(\forall x.\ ?P\ x \wedge ?Q\ x) \equiv (\forall x.\ ?P\ x) \wedge (\forall x.\ ?Q\ x)$ or its symmetric form, since the *rhs* is also a higher-order pattern.
3. Physical first-order patterns over raw λ-term structure without $\alpha\beta\eta$-equality; abstractions and bound variables are treated like quasi-constant term material.
   For example, the rule $?f\ ?x \in range\ ?f = True$ rewrites the term $g\ a \in range\ g$ to *True*, but will fail to match $g\ (h\ b) \in range\ (\lambda x.\ g\ (h\ x))$. However, offending subterms (in our case $?f\ ?x$, which is not a pattern) can be replaced by adding new variables and conditions like this: $?y = ?f\ ?x \Longrightarrow ?y \in range\ ?f = True$ is acceptable as a conditional rewrite rule of the second category since conditions can be arbitrary terms.

split declares case split rules.

cong declares congruence rules to the Simplifier context.

Congruence rules are equalities of the form

$$\ldots \Longrightarrow f\ ?x_1\ \ldots\ ?x_n = f\ ?y_1\ \ldots\ ?y_n$$

This controls the simplification of the arguments of f. For example, some arguments can be simplified under additional assumptions:

$$\begin{array}{l} ?P_1 \longleftrightarrow ?Q_1 \Longrightarrow \\ \quad (?Q_1 \Longrightarrow ?P_2 \longleftrightarrow ?Q_2) \Longrightarrow \\ \quad (?P_1 \longrightarrow ?P_2) \longleftrightarrow (?Q_1 \longrightarrow ?Q_2) \end{array}$$

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 $x = 0 \longrightarrow y + x = y$.

The following congruence rule for bounded quantifiers also supplies contextual information — about the bound variable:

$$\begin{array}{l} (?A = ?B) \Longrightarrow \\ \quad (\bigwedge x.\ x \in ?B \Longrightarrow ?P\ x \longleftrightarrow ?Q\ x) \Longrightarrow \\ \quad (\forall x \in ?A.\ ?P\ x) \longleftrightarrow (\forall x \in ?B.\ ?Q\ x) \end{array}$$

This congruence rule for conditional expressions can supply contextual information for simplifying the arms:

$$\begin{array}{l} ?p = ?q \Longrightarrow \\ \quad (?q \Longrightarrow ?a = ?c) \Longrightarrow \\ \quad (\neg\ ?q \Longrightarrow ?b = ?d) \Longrightarrow \\ \quad (\mathit{if}\ ?p\ \mathit{then}\ ?a\ \mathit{else}\ ?b) = (\mathit{if}\ ?q\ \mathit{then}\ ?c\ \mathit{else}\ ?d) \end{array}$$

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:

$$\begin{array}{l} ?p = ?q \Longrightarrow \\ \quad (\mathit{if}\ ?p\ \mathit{then}\ ?a\ \mathit{else}\ ?b) = (\mathit{if}\ ?q\ \mathit{then}\ ?a\ \mathit{else}\ ?b) \end{array}$$

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.

print_simpset prints the collection of rules declared to the Simplifier, which is also known as "simpset" internally; the "!" option indicates extra verbosity.

The implicit simpset of the theory context is propagated monotonically through the theory hierarchy: forming a new theory, the union of the simpsets of its imports are taken as starting point. Also note that definitional packages like **datatype**, **primrec**, **fun** routinely declare Simplifier rules to the target context, while plain **definition** is an exception in *not* declaring anything.

It is up the user to manipulate the current simpset further by explicitly adding or deleting theorems as simplification rules, or installing other tools via simplification procedures (§9.3.5). Good simpsets are hard to design. Rules that obviously simplify, like $?n + 0 \equiv ?n$ are good candidates for the implicit simpset, unless a special non-normalizing behavior of certain operations is intended. More specific rules (such as distributive laws, which duplicate subterms) should be added only for specific proof steps. Conversely, sometimes a rule needs to be deleted just for some part of a proof. The need of frequent additions or deletions may indicate a poorly designed simpset.

! The union of simpsets from theory imports (as described above) is not always a good starting point for the new theory. If some ancestors have deleted simplification rules because they are no longer wanted, while others have left those rules in, then the union will contain the unwanted rules, and thus have to be deleted again in the theory body.

9.3.3 Ordered rewriting with permutative rules

A rewrite rule is *permutative* if the left-hand side and right-hand side are the equal up to renaming of variables. The most common permutative rule is commutativity: $?x + ?y = ?y + ?x$. Other examples include $(?x - ?y) - ?z = (?x - ?z) - ?y$ in arithmetic and $insert\ ?x\ (insert\ ?y\ ?A) = insert\ ?y\ (insert\ ?x\ ?A)$ for sets. Such rules are common enough to merit special attention.

Because ordinary rewriting loops given such rules, the Simplifier employs a special strategy, called *ordered rewriting*. Permutative rules are detected and only applied if the rewriting step decreases the redex wrt. a given term ordering. For example, commutativity rewrites $b + a$ to $a + b$, but then

stops, because the redex cannot be decreased further in the sense of the term ordering.
The default is lexicographic ordering of term structure, but this could be also changed locally for special applications via `Simplifier.set_term_ord` in Isabelle/ML.

Permutative rewrite rules are declared to the Simplifier just like other rewrite rules. Their special status is recognized automatically, and their application is guarded by the term ordering accordingly.

Rewriting with AC operators

Ordered rewriting is particularly effective in the case of associative-commutative operators. (Associativity by itself is not permutative.) When dealing with an AC-operator f, keep the following points in mind:

- The associative law must always be oriented from left to right, namely $f\ (f\ x\ y)\ z = f\ x\ (f\ y\ z)$. The opposite orientation, if used with commutativity, leads to looping in conjunction with the standard term order.
- To complete your set of rewrite rules, you must add not just associativity (A) and commutativity (C) but also a derived rule *left-commutativity* (LC): $f\ x\ (f\ y\ z) = f\ y\ (f\ x\ z)$.

Ordered rewriting with the combination of A, C, and LC sorts a term lexicographically — the rewriting engine imitates bubble-sort.

experiment
  fixes $f :: 'a \Rightarrow 'a \Rightarrow 'a$ (**infix** $\langle\cdot\rangle$ 60)
  assumes *assoc*: $(x \cdot y) \cdot z = x \cdot (y \cdot z)$
  assumes *commute*: $x \cdot y = y \cdot x$
begin

lemma *left_commute*: $x \cdot (y \cdot z) = y \cdot (x \cdot z)$
proof −
  have $(x \cdot y) \cdot z = (y \cdot x) \cdot z$ **by** (*simp only*: *commute*)
  then show *?thesis* **by** (*simp only*: *assoc*)
qed

lemmas *AC_rules* = *assoc commute left_commute*

Thus the Simplifier is able to establish equalities with arbitrary permutations of subterms, by normalizing to a common standard form. For example:

lemma $(b \cdot c) \cdot a = xxx$
apply (*simp only*: *AC_rules*)

1. $a \cdot (b \cdot c) = xxx$

oops

lemma $(b \cdot c) \cdot a = a \cdot (b \cdot c)$ **by** (*simp only*: *AC_rules*)
lemma $(b \cdot c) \cdot a = c \cdot (b \cdot a)$ **by** (*simp only*: *AC_rules*)
lemma $(b \cdot c) \cdot a = (c \cdot b) \cdot a$ **by** (*simp only*: *AC_rules*)

end

Martin and Nipkow [31] discuss the theory and give many examples; other algebraic structures are amenable to ordered rewriting, such as Boolean rings. The Boyer-Moore theorem prover [11] also employs ordered rewriting.

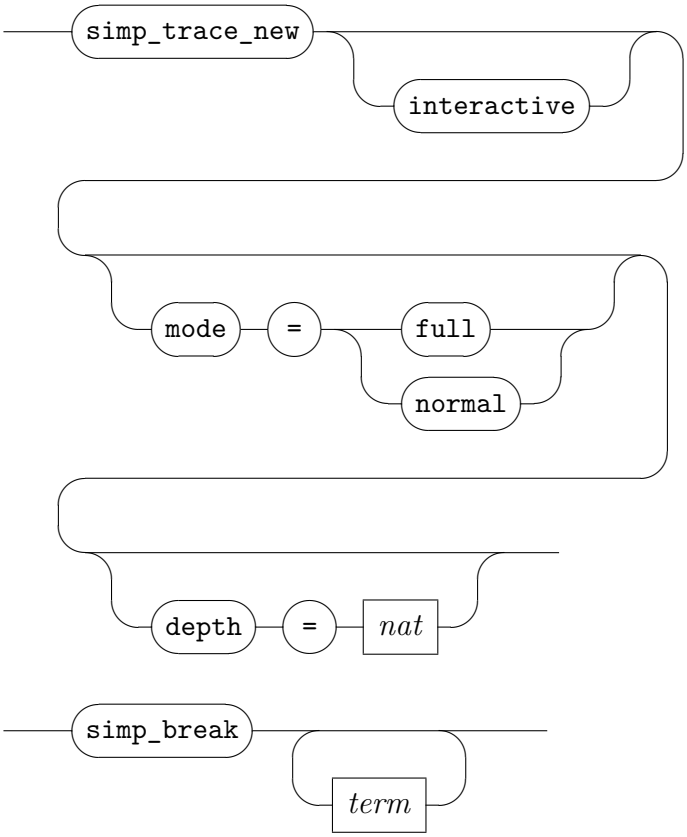
Re-orienting equalities

Another application of ordered rewriting uses the derived rule *eq_commute*: $(?a = ?b) = (?b = ?a)$ to reverse equations.

This is occasionally useful to re-orient local assumptions according to the term ordering, when other built-in mechanisms of reorientation and mutual simplification fail to apply.

9.3.4 Simplifier tracing and debugging

simp_trace	:	*attribute*	default *false*
simp_trace_depth_limit	:	*attribute*	default 1
simp_debug	:	*attribute*	default *false*
simp_trace_new	:	*attribute*	
simp_break	:	*attribute*	



These attributes and configurations options control various aspects of Simplifier tracing and debugging.

simp_trace makes the Simplifier output internal operations. This includes rewrite steps (but not traces from simproc calls), but also bookkeeping like modifications of the simpset.

simp_trace_depth_limit limits the effect of *simp_trace* to the given depth of recursive Simplifier invocations (when solving conditions of rewrite rules).

simp_debug makes the Simplifier output some extra information about internal operations. This includes any attempted invocation of simplification procedures and the corresponding traces.

simp_trace_new controls Simplifier tracing within Isabelle/PIDE applications, notably Isabelle/jEdit [56]. This provides a hierarchical representation of the rewriting steps performed by the Simplifier.

Users can configure the behaviour by specifying breakpoints, verbosity and enabling or disabling the interactive mode. In normal verbosity (the default), only rule applications matching a breakpoint will be shown to the user. In full verbosity, all rule applications will be logged. Interactive mode interrupts the normal flow of the Simplifier and defers the decision how to continue to the user via some GUI dialog.

simp_break declares term or theorem breakpoints for *simp_trace_new* as described above. Term breakpoints are patterns which are checked for matches on the redex of a rule application. Theorem breakpoints trigger when the corresponding theorem is applied in a rewrite step. For example:

declare *conjI* [*simp_break*]
declare [[*simp_break ?x* ∧ *?y*]]

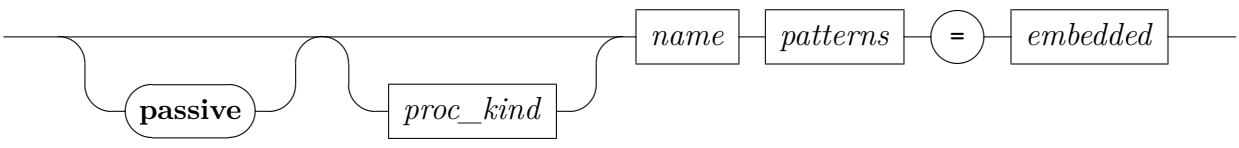
9.3.5 Simplification procedures

A *simplification procedure* or *simproc* is an ML function that produces proven rewrite rules on demand. Simprocs are guarded by multiple *patterns* for 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. The function may choose to succeed with a specific result for the redex, or fail.

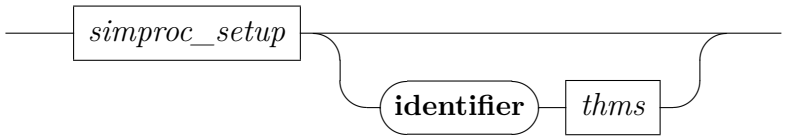
The successful result of a simproc needs to be a (possibly conditional) rewrite rule $t \equiv u$ that is applicable to the current redex. The rule will be applied just as any ordinary rewrite rule. It is expected to be already in *internal form* of the Pure logic, bypassing the automatic preprocessing of object-level equivalences.

simproc_setup : *local_theory* → *local_theory*
simproc_setup : *ML antiquotation*
simproc : *attribute*

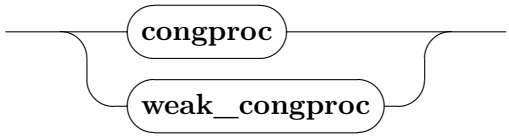
simproc_setup



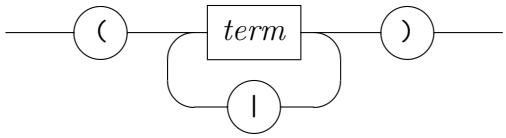
simproc_setup_id

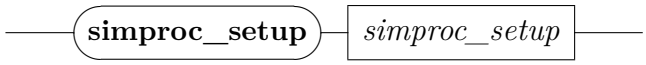


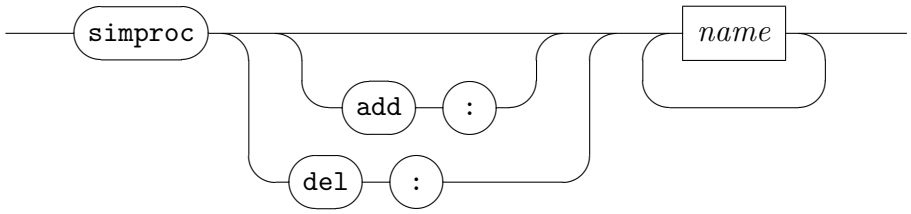
proc_kind



patterns







Command **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 embedded ML source, needs to be of type `morphism -> Proof.context -> 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); `NONE` indicates failure. The `Proof.context` argument holds the full context of the current Simplifier invocation.

The `morphism` tells how to move from the abstract context of the original definition into the concrete context of applications. This is only relevant for simprocs that are defined "**in**" a local theory context (e.g. **locale** with later **interpretation**).

By default, the simproc is declared to the current Simplifier context and thus *active*. The keyword **passive** avoids that: it merely defines a simproc that can be activated in a different context later on.

Regular simprocs produce rewrite rules on the fly, but it is also possible to congruence rules via the *proc_kind* keywords: **congproc** or **weak_congproc**. See also `~~/src/HOL/Imperative_HOL/ex/Congproc_Ex.thy` for further explanations and examples.

ML antiquotation *simproc_setup* is like command **simproc_setup**, with slightly extended syntax following *simproc_setup_id*. It allows to introduce a new simproc conveniently within an ML module, and refer directly to its ML value. For example, see various uses in `~~/src/HOL/Tools/record.ML`.

The optional **identifier** specifies characteristic theorems to distinguish simproc instances after application of morphisms, e.g. **locale** with multiple **interpretation**. See also the minimal example below.

Attributes [*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 by default, unless it specified as **passive**.

Examples

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") =
  ‹K (K (fn ct =>
    if HOLogic.is_unit (Thm.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.

The subsequent example shows how to define a local simproc with extra identifier to distinguish its instances after interpretation:

locale *loc* = **fixes** *x y* :: *'a* **assumes** *eq*: *x* = *y*

begin

ML ‹
```
  simproc_setup ‹proc (x) =
    ‹fn phi => fn _ => fn _ => SOME (Morphism.thm phi @{thm eq})›
    identifier loc_axioms›
›
```

end

9.3.6 Configurable Simplifier strategies

The core term-rewriting engine of the Simplifier is normally used in combination with some add-on components that modify the strategy and allow to integrate other non-Simplifier proof tools. These may be reconfigured in ML as explained below. Even if the default strategies of object-logics like Isabelle/HOL are used unchanged, it helps to understand how the standard Simplifier strategies work.

The subgoaler

```
Simplifier.set_subgoaler: (Proof.context -> int -> tactic) ->
  Proof.context -> Proof.context
Simplifier.prems_of: Proof.context -> thm list
```

The subgoaler is the tactic used to solve subgoals arising out of conditional rewrite rules or congruence rules. The default should be simplification itself. In rare situations, this strategy may need to be changed. For example, if the premise of a conditional rule is an instance of its conclusion, as in $Suc\ ?m < ?n \Longrightarrow ?m < ?n$, the default strategy could loop.

`Simplifier.set_subgoaler` *tac ctxt* sets the subgoaler of the context to *tac*. The tactic will be applied to the context of the running Simplifier instance.

`Simplifier.prems_of` *ctxt* retrieves the current set of premises from the context. This may be non-empty only if the Simplifier has been told to utilize local assumptions in the first place (cf. the options in §9.3.1).

As an example, consider the following alternative subgoaler:

ML_val ‹
```
  fun subgoaler_tac ctxt =
```

```
    assume_tac ctxt ORELSE'
    resolve_tac ctxt (Simplifier.prems_of ctxt) ORELSE'
    asm_simp_tac ctxt
>
```

This tactic first tries to solve the subgoal by assumption or by resolving with with one of the premises, calling simplification only if that fails.

The solver

```
type solver
Simplifier.mk_solver: string ->
  (Proof.context -> int -> tactic) -> solver
Simplifier.set_safe_solver:
  solver -> Proof.context -> Proof.context
Simplifier.add_safe_solver:
  solver -> Proof.context -> Proof.context
Simplifier.set_unsafe_solver:
  solver -> Proof.context -> Proof.context
Simplifier.add_unsafe_solver:
  solver -> Proof.context -> Proof.context
```

A solver is a tactic that attempts to solve a subgoal after simplification. Its core functionality is to prove trivial subgoals such as *True* and $t = t$, but object-logics might be more ambitious. For example, Isabelle/HOL performs a restricted version of linear arithmetic here.

Solvers are packaged up in abstract type `solver`, with `Simplifier.mk_solver` as the only operation to create a solver.

Rewriting does not instantiate unknowns. For example, rewriting alone cannot prove $a \in ?A$ since this requires instantiating *?A*. The solver, however, is an arbitrary tactic and may instantiate unknowns as it pleases. This is the only way the Simplifier can handle a conditional rewrite rule whose condition contains extra variables. When a simplification tactic is to be combined with other provers, especially with the Classical Reasoner, it is important whether it can be considered safe or not. For this reason a simpset contains two solvers: safe and unsafe.

The standard simplification strategy solely uses the unsafe solver, which is appropriate in most cases. For special applications where the simplification process is not allowed to instantiate unknowns within the goal, simplification starts with the safe solver, but may still apply the ordinary unsafe one in nested simplifications for conditional rules or congruences. Note that in this

way the overall tactic is not totally safe: it may instantiate unknowns that appear also in other subgoals.

`Simplifier.mk_solver` *name tac* turns *tac* into a solver; the *name* is only attached as a comment and has no further significance.

`Simplifier.set_safe_solver` *solver ctxt* installs *solver* as the safe solver of *ctxt*.

`Simplifier.add_safe_solver` *solver ctxt* adds *solver* as an additional safe solver; it will be tried after the solvers which had already been present in *ctxt*.

`Simplifier.set_unsafe_solver` *solver ctxt* installs *solver* as the unsafe solver of *ctxt*.

`Simplifier.add_unsafe_solver` *solver ctxt* adds *solver* as an additional unsafe solver; it will be tried after the solvers which had already been present in *ctxt*.

The solver tactic is invoked with the context of the running Simplifier. Further operations may be used to retrieve relevant information, such as the list of local Simplifier premises via `Simplifier.prems_of` — this list may be non-empty only if the Simplifier runs in a mode that utilizes local assumptions (see also §9.3.1). The solver is also presented the full goal including its assumptions in any case. Thus it can use these (e.g. by calling `assume_tac`), even if the Simplifier proper happens to ignore local premises at the moment.

As explained before, the subgoaler is also used to solve the premises of congruence rules. These are usually of the form $s = ?x$, where s needs to be simplified and $?x$ needs to be instantiated with the result. Typically, the subgoaler will invoke the Simplifier at some point, which will eventually call the solver. For this reason, solver tactics must be prepared to solve goals of the form $t = ?x$, usually by reflexivity. In particular, reflexivity should be tried before any of the fancy automated proof tools.

It may even happen that due to simplification the subgoal is no longer an equality. For example, $False \longleftrightarrow ?Q$ could be rewritten to $\neg\ ?Q$. To cover this case, the solver could try resolving with the theorem $\neg\ False$ of the object-logic.

! If a premise of a congruence rule cannot be proved, then the congruence is ignored. This should only happen if the rule is *conditional* — that is, contains premises not of the form $t = ?x$. Otherwise it indicates that some congruence rule, or possibly the subgoaler or solver, is faulty.

The looper

```
Simplifier.set_loop: (Proof.context -> int -> tactic) ->
  Proof.context -> Proof.context
Simplifier.add_loop: string * (Proof.context -> int -> tactic) ->
  Proof.context -> Proof.context
Simplifier.del_loop: string -> Proof.context -> Proof.context
Splitter.add_split: thm -> Proof.context -> Proof.context
Splitter.add_split: thm -> Proof.context -> Proof.context
Splitter.add_split_bang:
  thm -> Proof.context -> Proof.context
Splitter.del_split: thm -> Proof.context -> Proof.context
```

The looper is a list of tactics that are applied after simplification, in case the solver failed to solve the simplified goal. If the looper succeeds, the simplification process is started all over again. Each of the subgoals generated by the looper is attacked in turn, in reverse order.

A typical looper is *case splitting*: the expansion of a conditional. Another possibility is to apply an elimination rule on the assumptions. More adventurous loopers could start an induction.

`Simplifier.set_loop` *tac ctxt* installs *tac* as the only looper tactic of *ctxt*.

`Simplifier.add_loop` (*name*, *tac*) *ctxt* adds *tac* as an additional looper tactic with name *name*, which is significant for managing the collection of loopers. The tactic will be tried after the looper tactics that had already been present in *ctxt*.

`Simplifier.del_loop` *name ctxt* deletes the looper tactic that was associated with *name* from *ctxt*.

`Splitter.add_split` *thm ctxt* adds split tactic for *thm* as additional looper tactic of *ctxt* (overwriting previous split tactic for the same constant).

`Splitter.add_split_bang` *thm ctxt* adds aggressive (see §9.3.1) split tactic for *thm* as additional looper tactic of *ctxt* (overwriting previous split tactic for the same constant).

`Splitter.del_split` *thm ctxt* deletes the split tactic corresponding to *thm* from the looper tactics of *ctxt*.

The splitter replaces applications of a given function; the right-hand side of the replacement can be anything. For example, here is a splitting rule for conditional expressions:

$?P\ (\mathit{if}\ ?Q\ ?x\ ?y) \longleftrightarrow (?Q \longrightarrow ?P\ ?x) \wedge (\neg\ ?Q \longrightarrow ?P\ ?y)$

Another example is the elimination operator for Cartesian products (which happens to be called *case_prod* in Isabelle/HOL:

$?P\ (\mathit{case_prod}\ ?f\ ?p) \longleftrightarrow (\forall\, a\ b.\ ?p = (a,\ b) \longrightarrow ?P\ (f\ a\ b))$

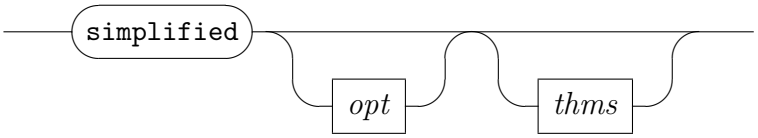
For technical reasons, there is a distinction between case splitting in the conclusion and in the premises of a subgoal. The former is done by `Splitter.split_tac` with rules like *if_split* or *option.split*, which do not split the subgoal, while the latter is done by `Splitter.split_asm_tac` with rules like *if_split_asm* or *option.split_asm*, which split the subgoal. The function `Splitter.add_split` automatically takes care of which tactic to call, analyzing the form of the rules given as argument; it is the same operation behind *split* attribute or method modifier syntax in the Isar source language.

Case splits should be allowed only when necessary; they are expensive and hard to control. Case-splitting on if-expressions in the conclusion is usually beneficial, so it is enabled by default in Isabelle/HOL and Isabelle/FOL/ZF.

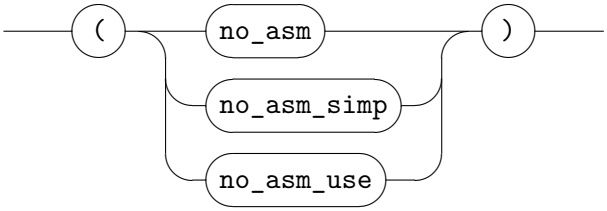
! With `Splitter.split_asm_tac` as looper component, the Simplifier may split subgoals! This might cause unexpected problems in tactic expressions that silently assume 0 or 1 subgoals after simplification.

9.3.7 Forward simplification

simplified : *attribute*



opt



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 *no_asm* etc. tune the Simplifier in the same way as the for the *simp* method.

Note that forward simplification restricts the Simplifier to its most basic operation of term rewriting; solver and looper tactics (§9.3.6) are *not* involved here. The *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 [48] problems 40 and 41 in a few milliseconds (including full proof reconstruction):

lemma $(\exists\, y.\ \forall\, x.\ F\ x\ y \longleftrightarrow F\ x\ x) \longrightarrow \neg\ (\forall\, x.\ \exists\, y.\ \forall\, z.\ F\ z\ y \longleftrightarrow \neg\ F\ z\ x)$
by *blast*

lemma $(\forall\, z.\ \exists\, y.\ \forall\, x.\ f\ x\ y \longleftrightarrow f\ x\ z \wedge \neg\ f\ x\ x) \longrightarrow \neg\ (\exists\, z.\ \forall\, x.\ f\ x\ z)$
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 Γ and Δ are sets of formulae.[3] The sequent $P_1, \ldots, P_m \vdash Q_1, \ldots, Q_n$ is **valid** if $P_1 \wedge \ldots \wedge P_m$ implies $Q_1 \vee \ldots \vee Q_n$. 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 $(\longrightarrow I)$ is the rule

$$\frac{P, \Gamma \vdash \Delta, Q}{\Gamma \vdash \Delta, P \longrightarrow Q} \, (\longrightarrow R)$$

Applying the rule backwards, this breaks down some implication on the right side of a sequent; Γ and Δ stand for the sets of formulae that are unaffected by the inference. The analogue of the pair $(\vee I1)$ and $(\vee I2)$ is the single rule

$$\frac{\Gamma \vdash \Delta, P, Q}{\Gamma \vdash \Delta, P \vee Q} \, (\vee R)$$

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 $(P \longrightarrow Q) \vee (Q \longrightarrow P)$. Working backwards, we reduce this

[3] For first-order logic, sequents can equivalently be made from lists or multisets of formulae.

formula to a basic sequent:

$$\dfrac{\dfrac{\dfrac{P,\ Q \vdash Q,\ P}{P \vdash Q,\ (Q \longrightarrow P)}\ (\longrightarrow R)}{\vdash (P \longrightarrow Q),\ (Q \longrightarrow P)}\ (\longrightarrow R)}{\vdash (P \longrightarrow Q) \vee (Q \longrightarrow P)}\ (\vee R)$$

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 [47, 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 $P_1 \Longrightarrow \ldots \Longrightarrow P_m \Longrightarrow \neg\ Q_2 \Longrightarrow \ldots \Longrightarrow \neg\ Q_n \Longrightarrow Q_1$ 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 $(\vee E)$, $(\bot E)$ 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 $(\wedge E1, 2)$ and $(\forall\, E)$. But note that the rule $(\neg L)$ has no effect under our representation of sequents!

$$\dfrac{\Gamma \vdash \Delta,\ P}{\neg\ P,\ \Gamma \vdash \Delta}\ (\neg L)$$

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: $\neg\ P \Longrightarrow (\neg\ R \Longrightarrow P) \Longrightarrow R$

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 $(\wedge I)$, which might be called $(\neg\wedge E)$, is

$$\neg\ (P \wedge Q) \Longrightarrow (\neg\ R \Longrightarrow P) \Longrightarrow (\neg\ R \Longrightarrow Q) \Longrightarrow R$$

Similarly, the swapped form of $(\longrightarrow I)$ is

$$\neg\,(P \longrightarrow Q) \Longrightarrow (\neg\,R \Longrightarrow P \Longrightarrow Q) \Longrightarrow R$$

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

$$P_1 \Longrightarrow \ldots \Longrightarrow P_m \Longrightarrow \neg\,Q_1 \Longrightarrow \ldots \Longrightarrow \neg\,Q_n \Longrightarrow \bot$$

Extra rules for the sequent calculus

As mentioned, destruction rules such as $(\wedge E1,\, 2)$ and $(\forall\, E)$ must be replaced by sequent-style elimination rules. In addition, we need rules to embody the classical equivalence between $P \longrightarrow Q$ and $\neg\,P \vee Q$. The introduction rules $(\vee I1,\, 2)$ are replaced by a rule that simulates $(\vee R)$:

$$(\neg\,Q \Longrightarrow P) \Longrightarrow P \vee Q$$

The destruction rule $(\longrightarrow E)$ is replaced by

$$(P \longrightarrow Q) \Longrightarrow (\neg\,P \Longrightarrow R) \Longrightarrow (Q \Longrightarrow R) \Longrightarrow R$$

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:

$$\frac{\Gamma \vdash \Delta,\, \exists\, x.\ P\ x,\, P\ t}{\Gamma \vdash \Delta,\, \exists\, x.\ P\ x}\ (\exists\, R) \qquad \frac{P\ t,\, \forall\, x.\ P\ x,\, \Gamma \vdash \Delta}{\forall\, x.\ P\ x,\, \Gamma \vdash \Delta}\ (\forall\, L)$$

Thus both kinds of quantifier may be replicated. Theorems requiring multiple uses of a universal formula are easy to invent; consider

$$(\forall\, x.\ P\ x \longrightarrow P\ (f\ x)) \wedge P\ a \longrightarrow P\ (f^n\ a)$$

for any $n > 1$. Natural examples of the multiple use of an existential formula are rare; a standard one is $\exists\, x.\ \forall\, y.\ P\ x \longrightarrow P\ y$.

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:

$\forall x.\ P\ x \Longrightarrow (P\ t \Longrightarrow Q) \Longrightarrow Q$

Elim-resolution with this rule will delete the universal formula after a single use. To replicate universal quantifiers, replace the rule by

$\forall x.\ P\ x \Longrightarrow (P\ t \Longrightarrow \forall x.\ P\ x \Longrightarrow Q) \Longrightarrow Q$

To replicate existential quantifiers, replace $(\exists\, I)$ by

$(\neg\ (\exists x.\ P\ x) \Longrightarrow P\ t) \Longrightarrow \exists x.\ P\ x$

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

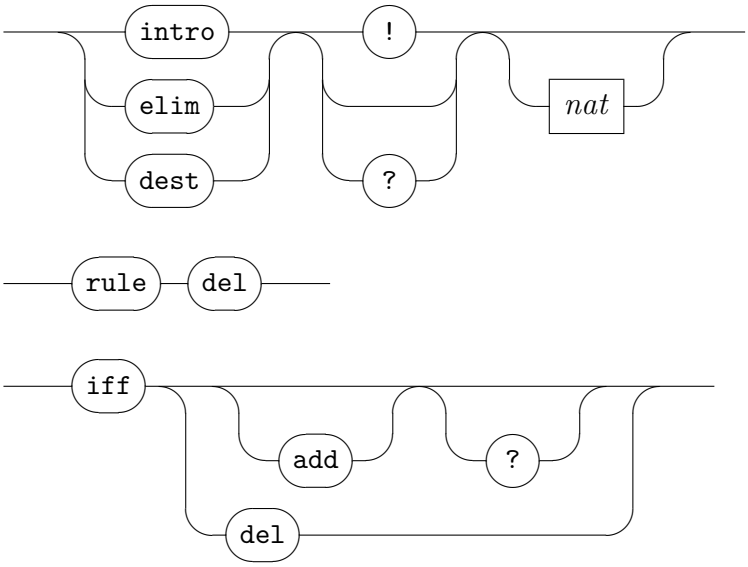
The proof tools of the Classical Reasoner depend on collections of rules declared in the context, which are classified as introduction, elimination or destruction and as *safe* or *unsafe*. In general, safe rules can be attempted blindly, while unsafe rules must be used with care. A safe rule must never reduce a provable goal to an unprovable set of subgoals.

The rule $P \Longrightarrow P \vee Q$ is unsafe because it reduces $P \vee Q$ to P, which might turn out as premature choice of an unprovable subgoal. Any rule whose premises contain new unknowns is unsafe. The elimination rule $\forall x.\ P\ x \Longrightarrow (P\ t \Longrightarrow Q) \Longrightarrow Q$ 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 $P\ t \Longrightarrow \exists x.\ P\ x$ is unsafe for similar reasons. The quantifier duplication rule $\forall x.\ P\ x \Longrightarrow (P\ t \Longrightarrow \forall x.\ P\ x \Longrightarrow Q) \Longrightarrow Q$ 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 $(\vee E)$ 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 that instantiates an unknown in the proof state is unsafe — thus matching must be used, rather than unification.

Even proof by assumption is unsafe if it instantiates unknowns shared with other subgoals.

print_claset*	:	*context* $\rightarrow$
intro	:	*attribute*
elim	:	*attribute*
dest	:	*attribute*
rule	:	*attribute*
iff	:	*attribute*
swapped	:	*attribute*



print_claset prints the collection of rules declared to the Classical Reasoner, i.e. the `claset` within the context.

intro, *elim*, and *dest* declare introduction, elimination, and destruction rules, respectively. By default, rules are considered as *unsafe* (i.e. not applied blindly without backtracking), while "!" classifies as *safe*. Rule declarations marked by "*?*" coincide with those of Isabelle/Pure, cf. §6.4.3 (i.e. are only applied in single steps of the *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. Otherwise, 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.

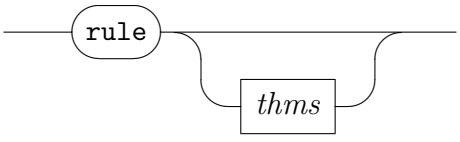
iff declares logical equivalences to the Simplifier and the Classical reasoner at the same time. Non-conditional rules result in a safe introduction and elimination pair; conditional ones are considered unsafe. Rules with negative conclusion are automatically inverted (using $\neg$-elimination internally).

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 $\neg\ P \Longrightarrow (\neg\ R \Longrightarrow P) \Longrightarrow R$ 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*



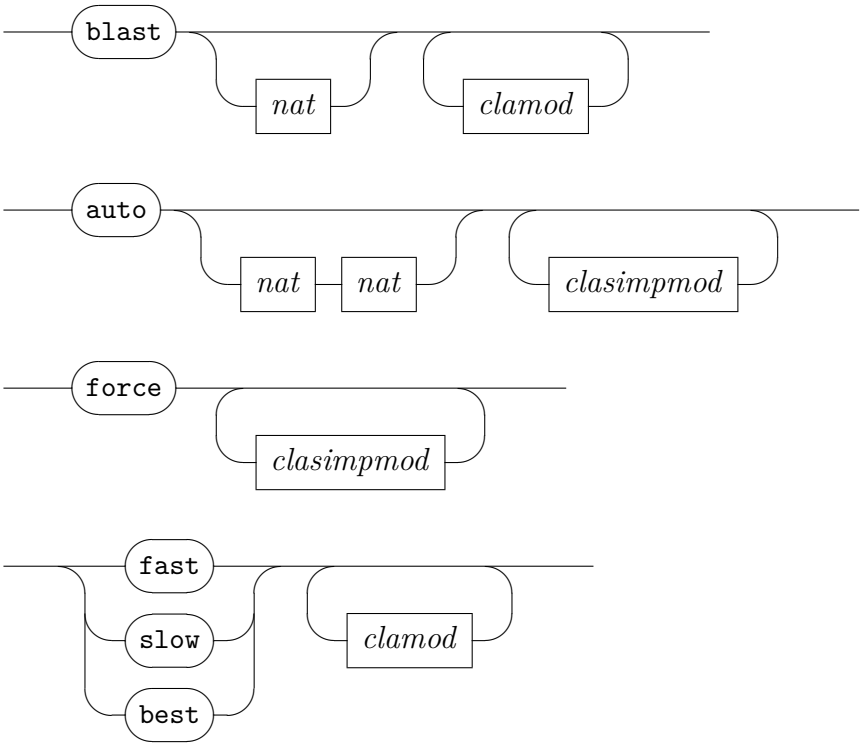
rule as offered by the Classical Reasoner is a refinement over the Pure one (see §6.4.3). Both versions work the same, but the classical version observes the classical rule context in addition to that of Isabelle/Pure.

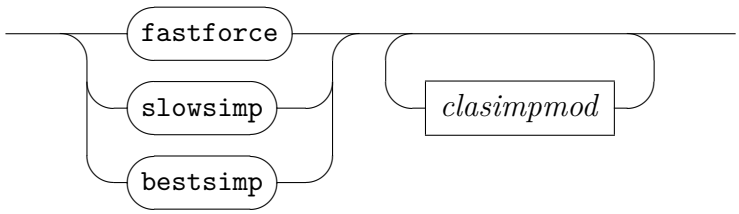
Common object logics (HOL, ZF, etc.) declare a rich collection of classical rules (even if these would qualify as intuitionistic ones), but only few declarations to the rule context of Isabelle/Pure (§6.4.3).

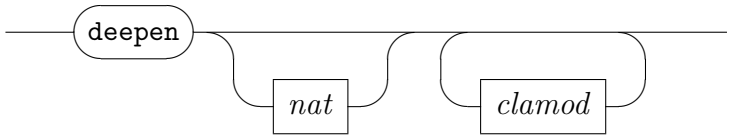
contradiction solves some goal by contradiction, deriving any result from both $\neg\ A$ and A. Chained facts, which are guaranteed to participate, may appear in either order.

9.4.4 Fully automated methods

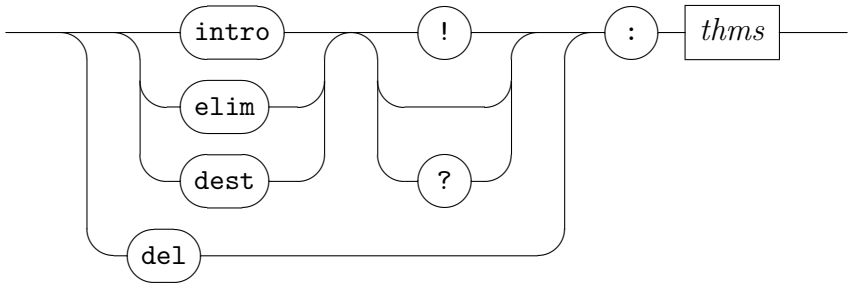
blast : *method*
auto : *method*
force : *method*
fast : *method*
slow : *method*
best : *method*
fastforce : *method*
slowsimp : *method*
bestsimp : *method*
deepen : *method*



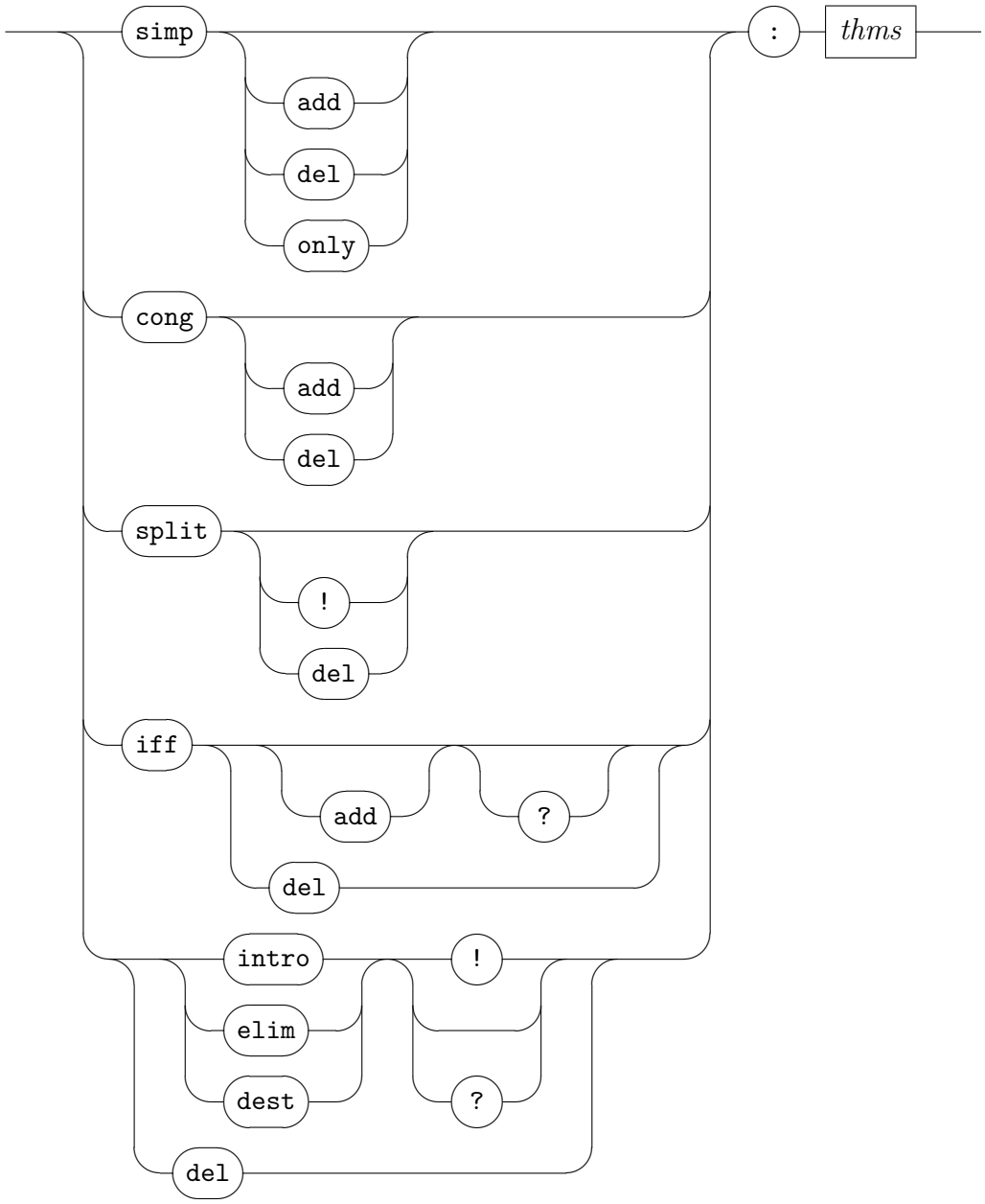

fastforce
slowsimp
bestsimp
clasimpmod


deepen
nat
clamod

clamod


intro
elim
dest
del
!
?
:
thms

clasimpmod



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 *?f ?x* $\in$ *range ?f* in HOL. There are often alternatives to such rules, for example *?b* = *?f ?x* $\Longrightarrow$ *?b* $\in$ *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 unknown'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 builtin 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 Simplifier as additional wrapper. The name *fastforce*, reflects the behaviour of this popular method better without requiring an understanding of its implementation.

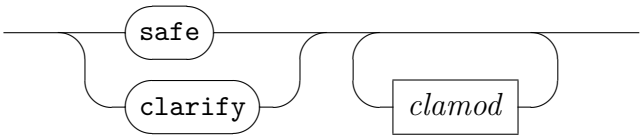
deepen works by exhaustive search up to a certain depth. The start depth is 4 (unless specified explicitly), and the depth is increased iteratively up to 10. Unsafe rules are modified to preserve the formula they act on, so that it be used repeatedly. This method can prove more goals than *fast*, but is much slower, for example if the assumptions have many universal quantifiers.

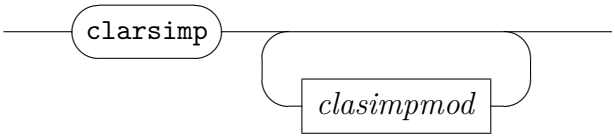
Any of the above methods support additional modifiers of the context of classical (and simplifier) rules, but the ones related to the Simplifier are explicitly prefixed by *simp* here. The semantics of these ad-hoc rule declarations is analogous to the attributes given before. Facts provided by forward chaining are inserted into the goal before commencing proof search.

9.4.5 Partially 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.

safe : *method*
clarify : *method*
clarsimp : *method*





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*.

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	:	*method*
inst_step	:	*method*
step	:	*method*
slow_step	:	*method*
clarify_step	:	*method*

These are the primitive tactics behind the automated proof methods of the Classical Reasoner. By calling them yourself, you can execute these procedures one step at a time.

safe_step performs a safe step on the first subgoal. 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 is like *safe_step*, but allows unknowns to be instantiated.

step is the basic step of the proof procedure, it operates on the first subgoal. The unsafe wrapper tacticals are applied to a tactic that tries *safe*, *inst_step*, or applies an unsafe rule from the context.

slow_step resembles *step*, but allows backtracking between using safe rules with instantiation (*inst_step*) and using unsafe rules. The resulting search space is larger.

clarify_step performs a safe step on the first subgoal; no splitting step is applied. For example, the subgoal $A \wedge B$ 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.

9.4.7 Modifying the search step

```
type wrapper = (int -> tactic) -> (int -> tactic)
infix addSWrapper: Proof.context *
  (string * (Proof.context -> wrapper)) -> Proof.context
infix addSbefore: Proof.context *
  (string * (Proof.context -> int -> tactic)) -> Proof.context
infix addSafter: Proof.context *
  (string * (Proof.context -> int -> tactic)) -> Proof.context
infix delSWrapper: Proof.context * string -> Proof.context
infix addWrapper: Proof.context *
  (string * (Proof.context -> wrapper)) -> Proof.context
infix addbefore: Proof.context *
  (string * (Proof.context -> int -> tactic)) -> Proof.context
infix addafter: Proof.context *
  (string * (Proof.context -> int -> tactic)) -> Proof.context
infix delWrapper: Proof.context * string -> Proof.context
addSss: Proof.context -> Proof.context
addss: Proof.context -> Proof.context
```

The proof strategy of the Classical Reasoner is simple. Perform as many safe inferences as possible; or else, apply certain safe rules, allowing instantiation of unknowns; or else, apply an unsafe rule. The tactics also eliminate assumptions of the form $x = t$ by substitution if they have been set up to do so. They may perform a form of Modus Ponens: if there are assumptions $P \longrightarrow Q$ and P, then replace $P \longrightarrow Q$ by Q.

The classical reasoning tools — except *blast* — allow to modify this basic proof strategy by applying two lists of arbitrary *wrapper tacticals* to it. The first wrapper list, which is considered to contain safe wrappers only, affects *safe_step* and all the tactics that call it. The second one, which may contain unsafe wrappers, affects the unsafe parts of *step*, *slow_step*, and the tactics that call them. A wrapper transforms each step of the search, for example by attempting other tactics before or after the original step tactic. All members of a wrapper list are applied in turn to the respective step tactic.

Initially the two wrapper lists are empty, which means no modification of the step tactics. Safe and unsafe wrappers are added to the context with the functions given below, supplying them with wrapper names. These names may be used to selectively delete wrappers.

ctxt addSWrapper (*name*, *wrapper*) adds a new wrapper, which should yield a safe tactic, to modify the existing safe step tactic.

ctxt addSbefore (*name*, *tac*) adds the given tactic as a safe wrapper, such that it is tried *before* each safe step of the search.

ctxt addSafter (*name*, *tac*) adds the given tactic as a safe wrapper, such that it is tried when a safe step of the search would fail.

ctxt delSWrapper name deletes the safe wrapper with the given name.

ctxt addWrapper (*name*, *wrapper*) adds a new wrapper to modify the existing (unsafe) step tactic.

ctxt addbefore (*name*, *tac*) adds the given tactic as an unsafe wrapper, such that it its result is concatenated *before* the result of each unsafe step.

ctxt addafter (*name*, *tac*) adds the given tactic as an unsafe wrapper, such that it its result is concatenated *after* the result of each unsafe step.

ctxt delWrapper name deletes the unsafe wrapper with the given name.

addSss adds the simpset of the context to its classical set. The assumptions and goal will be simplified, in a rather safe way, after each safe step of the search.

addss adds the simpset of the context to its classical set. The assumptions and goal will be simplified, before the each unsafe step of the search.

9.5 Object-logic setup

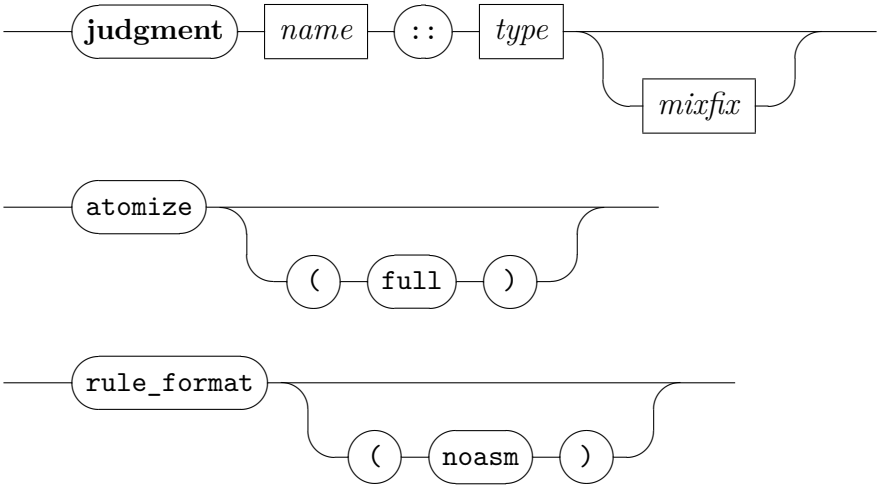
judgment : *theory* → *theory*
atomize : *method*
atomize : *attribute*
rule_format : *attribute*
rulify : *attribute*

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 *prop* that covers universal quantification $\bigwedge$ and implication $\Longrightarrow$).

Common object-logics are sufficiently expressive to internalize rule statements over $\bigwedge$ and $\Longrightarrow$ 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. $\bigwedge x.\ x \in A \Longrightarrow P\ x$ versus $\forall x \in A.\ P\ x$.

From the following language elements, only the *atomize* method and *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.



judgment $c :: \sigma\ (mx)$ declares constant c as the truth judgment of the current object-logic. Its type σ should specify a coercion of the category of object-level propositions to *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 $\bigwedge$, $\Longrightarrow$, and $\equiv$. Meta-level conjunction should be covered as well (this is particularly important for locales, see §5.7).

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 ($\longrightarrow$) by the corresponding rule statements over $\bigwedge$ and $\Longrightarrow$.

9.6 Tracing higher-order unification

unify_trace	:	*attribute*	default *false*
unify_trace_simp	:	*attribute*	default *false*
unify_trace_types	:	*attribute*	default *false*
unify_trace_bound	:	*attribute*	default 50
unify_search_bound	:	*attribute*	default 60

Higher-order unification works well in most practical situations, but sometimes needs extra care to identify problems. These tracing options may help.

unify_trace controls whether unify trace messages will be printed (controlled via more fine-grained tracing options below).

unify_trace_simp controls tracing of the simplification phase of higher-order unification.

unify_trace_types controls tracing of potential incompleteness, when unification is not considering all possible instantiations of schematic type variables.

unify_trace_bound determines the depth where unification starts to print tracing information once it reaches depth; 0 for full tracing. At the default value, tracing information is almost never printed in practice.

unify_search_bound prevents unification from searching past the given depth. Because of this bound, higher-order unification cannot return an infinite sequence, though it can return an exponentially long one. The search rarely approaches the default value in practice. If the search is cut off, unification prints a warning "Unification bound exceeded".

! Options for unification cannot be modified in a local context. Only the global theory content is taken into account.

Part III

Isabelle/HOL

Chapter 10

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 [20]. It extends Church's original logic [14] 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 [50].

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 $Pair\ a\ b$). 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.

Chapter 11

Derived specification elements

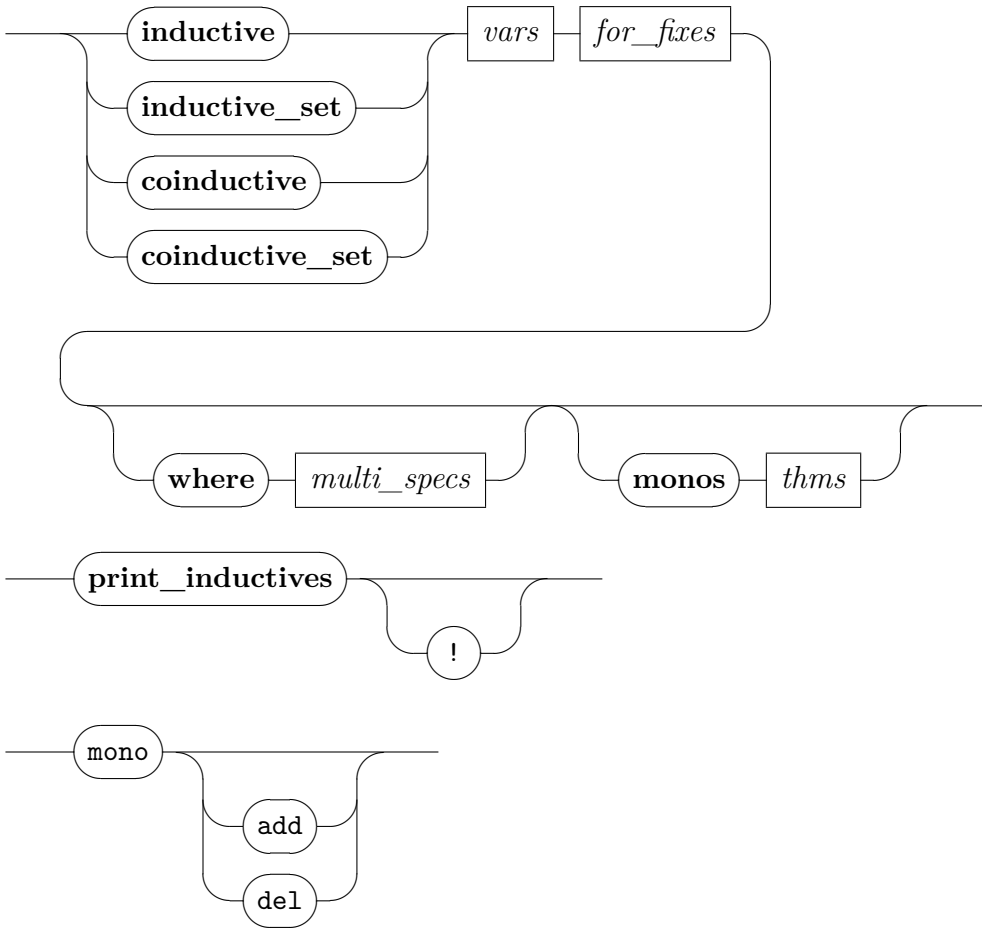
11.1 Inductive and coinductive definitions

inductive	:	*local_theory* → *local_theory*
inductive_set	:	*local_theory* → *local_theory*
coinductive	:	*local_theory* → *local_theory*
coinductive_set	:	*local_theory* → *local_theory*
print_inductives*	:	*context* →
mono	:	*attribute*

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 $A \leq B \Longrightarrow \mathcal{M}\ A \leq \mathcal{M}\ B$, 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 $\bigwedge/\Longrightarrow$ 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.

print_inductives prints (co)inductive definitions and monotonicity rules; the "!" option indicates extra verbosity.

mono declares monotonicity rules in the context. These rule are involved in the automated monotonicity proof of the above inductive and coinductive definitions.

11.1.1 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 $R_1_\ldots_R_n$*.intros*, 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*.

11.1.2 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 $A \leq B \Longrightarrow \mathcal{M}\ A \leq \mathcal{M}\ B$, 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 $(\ldots \longrightarrow \ldots) \Longrightarrow \ldots\ (\ldots \longrightarrow \ldots) \Longrightarrow \ldots \longrightarrow \ldots$. For example, in the case of the operator $\vee$, the corresponding theorem is

$$\frac{P_1 \longrightarrow Q_1 \quad P_2 \longrightarrow Q_2}{P_1 \vee P_2 \longrightarrow Q_1 \vee Q_2}$$

- De Morgan style equations for reasoning about the "polarity" of expressions, e.g.

$$\neg \neg P \longleftrightarrow P \qquad \neg (P \wedge Q) \longleftrightarrow \neg P \vee \neg Q$$

- Equations for reducing complex operators to more primitive ones whose monotonicity can easily be proved, e.g.

$$(P \longrightarrow Q) \longleftrightarrow \neg P \vee Q \qquad Ball\ A\ P \equiv \forall x.\ x \in A \longrightarrow P\ x$$

Examples

The finite powerset operator can be defined inductively like this:

```
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:

```
inductive acc :: ('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ bool
  for r :: 'a ⇒ 'a ⇒ bool  (infix ‹≺› 50)
where acc: (⋀y. y ≺ x ⟹ acc r y) ⟹ acc r x
```

Common logical connectives can be easily characterized as non-recursive inductive definitions with parameters, but without arguments.

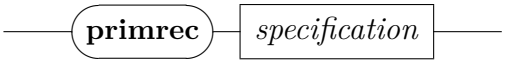
```
inductive AND for A B :: bool
where A ⟹ B ⟹ AND A B

inductive OR for A B :: bool
where A ⟹ OR A B
  | B ⟹ OR A B
```

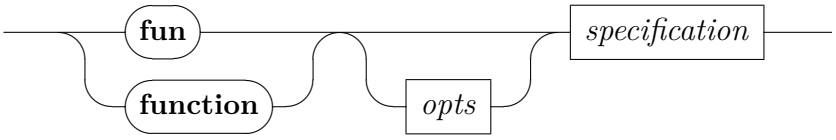
inductive *EXISTS* **for** $B :: 'a \Rightarrow bool$
where $B\ a \Longrightarrow EXISTS\ B$

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 [18] there is a hint that each connective can be characterized by its introductions, and the elimination can be constructed systematically.

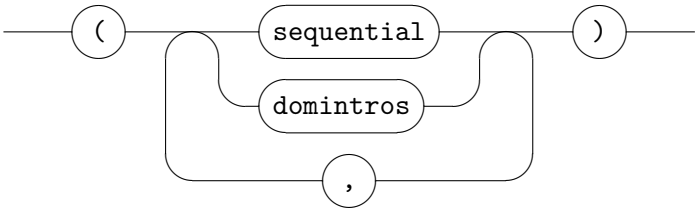
11.2 Recursive functions

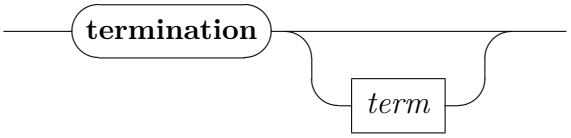
primrec : *local_theory* → *local_theory*
fun : *local_theory* → *local_theory*
function : *local_theory* → *proof*(*prove*)
termination : *local_theory* → *proof*(*prove*)
fun_cases : *local_theory* → *local_theory*

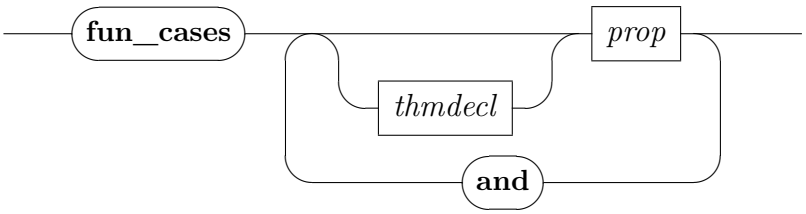




opts







primrec defines primitive recursive functions over datatypes (see also **datatype**). The given *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:

$$f\ x_1\ \ldots\ x_m\ (C\ y_1\ \ldots\ y_k)\ z_1\ \ldots\ z_n = rhs$$

such that C is a datatype constructor, *rhs* contains only the free variables on the left-hand side (or from the context), and all recursive occurrences of f in *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 *simp* by default, which enables standard proof methods like *simp* and *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 [25]. 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 simplification 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 automated proof attempts regarding pattern matching and termination. See [25] 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.

fun_cases generates specialized elimination rules for function equations. It expects one or more function equations and produces rules that eliminate the given equalities, following the cases given in the function definition.

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 theorem 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 simplification 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 **primrec** for evaluation functions that follow the same recursive structure.

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
```

Evaluation of arithmetic and boolean expressions

```
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)
```

Since the value of an expression depends on the value of its variables, the functions *evala* and *evalb* take an additional parameter, an *environment* that maps variables to their values.

Substitution on expressions can be defined similarly. The mapping f of type $'a \Rightarrow 'a\ aexp$ given as a parameter is lifted canonically on the types $'a\ aexp$ and $'a\ bexp$, respectively.

```
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
| substa f (Num n) = Num n
| substb f (Less a1 a2) = Less (substa f a1) (substa f a2)
| substb f (And b1 b2) = And (substb f b1) (substb f b2)
| substb f (Neg b) = Neg (substb f b)
```

In textbooks about semantics one often finds substitution theorems, which express the relationship between substitution and evaluation. For $'a\ aexp$

and *'a bexp*, we can prove such a theorem by mutual induction, followed by simplification.

lemma *subst_one*:
evala env (substa (Var (v := a')) a) = evala (env (v := evala env a')) a
evalb env (substb (Var (v := a')) b) = evalb (env (v := evala env a')) b
by (*induct a* **and** *b*) *simp_all*

lemma *subst_all*:
evala env (substa s a) = evala (λx. evala env (s x)) a
evalb env (substb s b) = evalb (λx. evala env (s x)) b
by (*induct a* **and** *b*) *simp_all*

Example: a substitution function for terms

Functions on datatypes with nested recursion are also defined by mutual primitive recursion.

datatype *('a, 'b) term = Var 'a | App 'b ('a, 'b) term list*

A substitution function on type *('a, 'b) term* can be defined as follows, by working simultaneously on *('a, 'b) term list*:

primrec *subst_term* :: *('a ⇒ ('a, 'b) term) ⇒ ('a, 'b) term ⇒ ('a, 'b) term* **and**
subst_term_list :: *('a ⇒ ('a, 'b) term) ⇒ ('a, 'b) term list ⇒ ('a, 'b) term list*
where
subst_term f (Var a) = f a
| *subst_term f (App b ts) = App b (subst_term_list f ts)*
| *subst_term_list f [] = []*
| *subst_term_list f (t # ts) = subst_term f t # subst_term_list f ts*

The recursion scheme follows the structure of the unfolded definition of type *('a, 'b) term*. To prove properties of this substitution function, mutual induction is needed:

lemma *subst_term (subst_term f1 ∘ f2) t =*
subst_term f1 (subst_term f2 t) **and**
subst_term_list (subst_term f1 ∘ f2) ts =
subst_term_list f1 (subst_term_list f2 ts)
by (*induct t* **and** *ts rule*: *subst_term.induct subst_term_list.induct*) *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* (λ*x*. *map_tree f* (*ts x*))

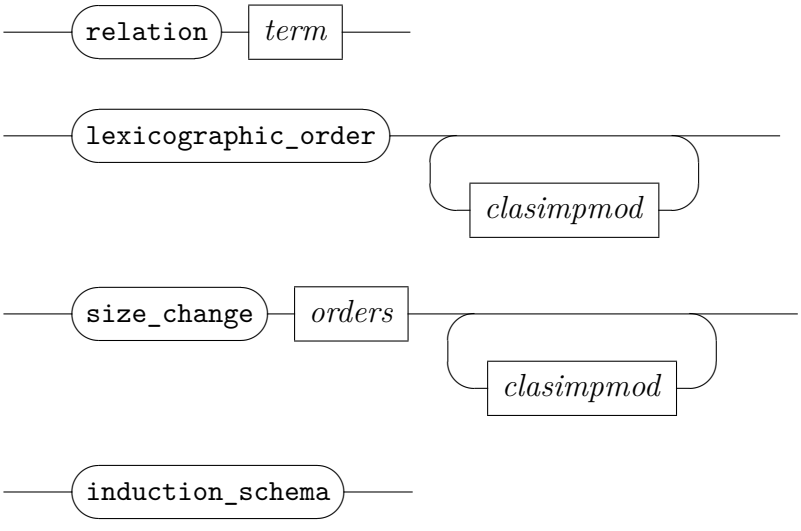
Note that all occurrences of functions such as *ts* above must be applied to an argument. In particular, *map_tree f* ∘ *ts* is not allowed here.

Here is a simple composition lemma for *map_tree*:

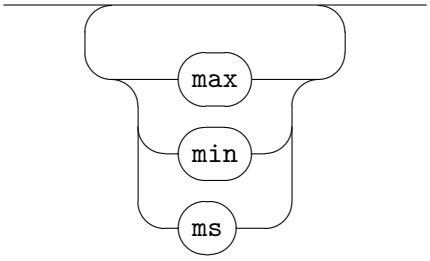
lemma *map_tree g* (*map_tree f t*) = *map_tree* (*g* ∘ *f*) *t*
by (*induct t*) *simp_all*

11.2.1 Proof methods related to recursive definitions

pat_completeness	:	*method*
relation	:	*method*
lexicographic_order	:	*method*
size_change	:	*method*
termination_simp	:	*attribute*
induction_schema	:	*method*



orders



pat_completeness is a specialized method to solve goals regarding the completeness of pattern matching, as required by the **function** package (cf. [25]).

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. [25]).

size_change also works on termination goals, using a variation of the size-change principle, together with a graph decomposition technique (see [26] 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*).

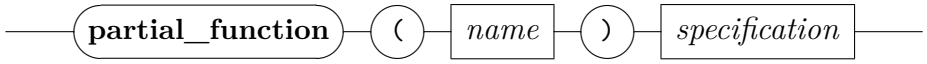
termination_simp declares extra rules for the simplifier, when invoked in termination proofs. This can be useful, e.g., for special rules involving size estimations.

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/Examples/Induction_Schema.thy` for examples.

11.2.2 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 *undefined* is returned by a recursive call, then the overall result must also be *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.

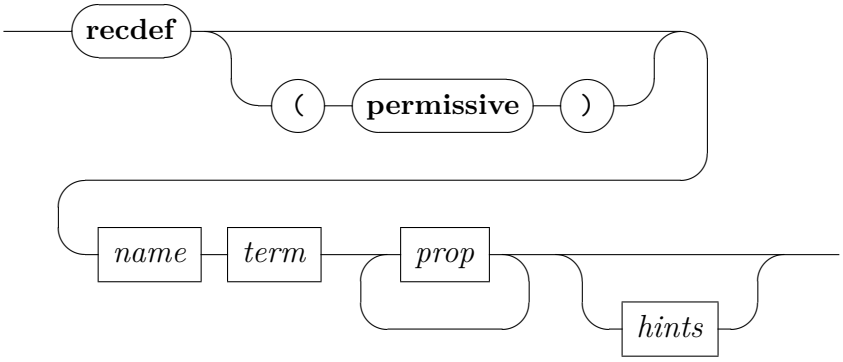
Experienced users may define new modes by instantiating the locale *partial_function_definitions* appropriately.

partial_function_mono declares rules for use in the internal monotonicity proofs of partial function definitions.

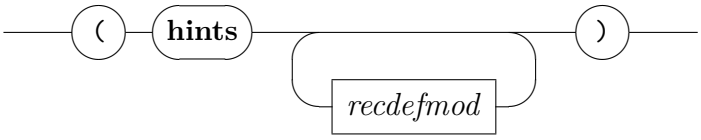
11.2.3 Old-style recursive function definitions (TFL)

recdef : *theory* → *theory*)

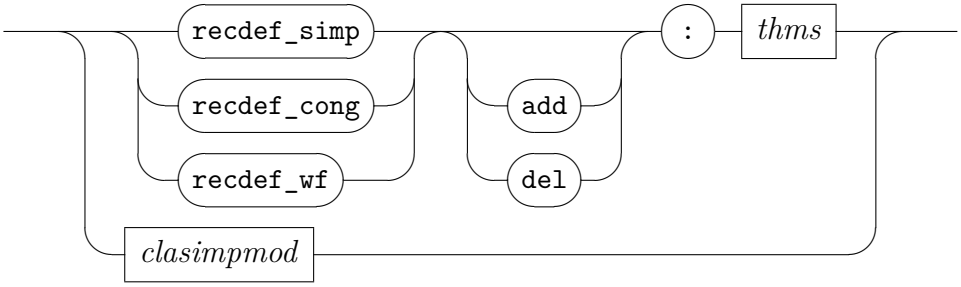
The old TFL command **recdef** for defining recursive is mostly obsolete; **function** or **fun** should be used instead.



hints



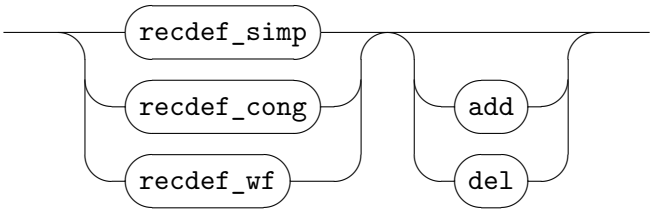
recdefmod



recdef defines general well-founded recursive functions (using the TFL package). The "(*permissive*)" option tells TFL to recover from failed proof attempts, returning unfinished results. The *recdef_simp*, *recdef_cong*, and *recdef_wf* hints refer to auxiliary rules to be used in the internal automated proof process of TFL. Additional *clasimpmod* declarations may be given to tune the context of the Simplifier (cf. §9.3) and Classical reasoner (cf. §9.4).

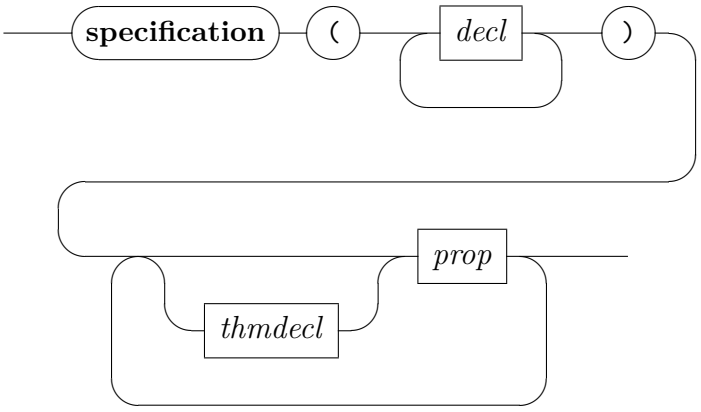
Hints for **recdef** may be also declared globally, using the following attributes.

recdef_simp : *attribute*
recdef_cong : *attribute*
recdef_wf : *attribute*

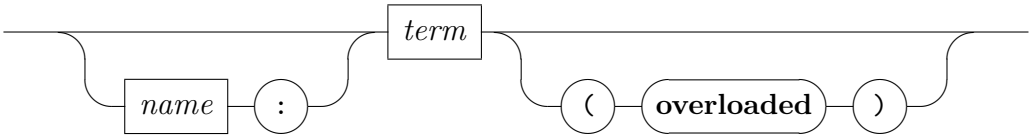


11.3 Definition by specification

specification : *theory* → *proof*(*prove*)



decl

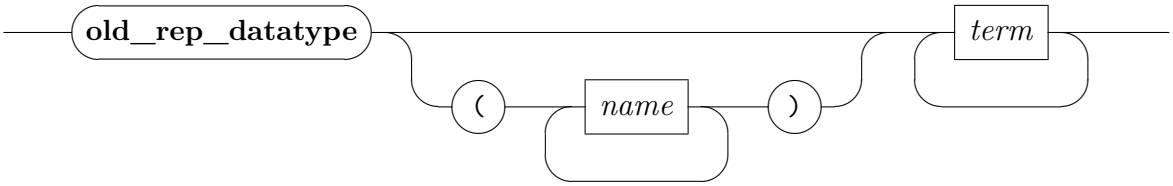


specification *decls* φ sets up a goal stating the existence of terms with the properties specified to hold for the constants given in *decls*. After finishing the proof, the theory will be augmented with definitions for the given constants, as well as with theorems stating the properties for these constants.

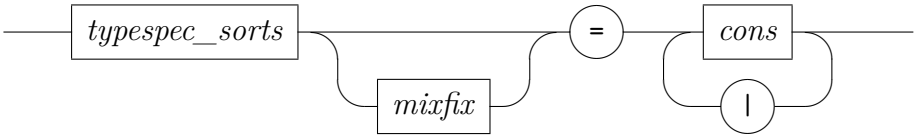
decl declares a constant to be defined by the specification given. The definition for the constant *c* is bound to the name *c_def* unless a theorem name is given in the declaration. Overloaded constants should be declared as such.

11.4 Old-style datatypes

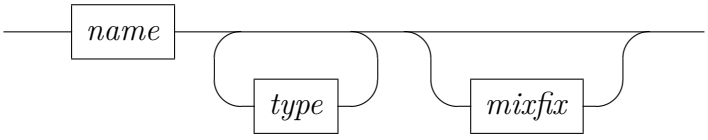
old_rep_datatype : *theory* → *proof*(*prove*)



spec



cons



old_rep_datatype represents existing types as old-style datatypes.

These commands are mostly obsolete; **datatype** should be used instead.

See [8] for more details on datatypes. Apart from proper proof methods for case analysis and induction, there are also emulations of ML tactics *case_tac* and *induct_tac* available, see §12.9; 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 *'a list* that is already in *Main*:

datatype *'a seq = Empty | Seq 'a 'a seq*

We can now prove some simple lemma by structural induction:

lemma *Seq x xs ≠ xs*
proof (*induct xs arbitrary*: *x*)
  case *Empty*

This case can be proved using the simplifier: the freeness properties of the datatype are already declared as *simp* rules.

  show *Seq x Empty ≠ Empty*
    by *simp*
next
  case (*Seq y ys*)

The step case is proved similarly.

  show *Seq x (Seq y ys) ≠ Seq y ys*
    using ‹*Seq y ys ≠ ys*› **by** *simp*
qed

Here is a more succinct version of the same proof:

lemma *Seq x xs ≠ xs*
  by (*induct xs arbitrary*: *x*) *simp_all*

11.5 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 [35] for more details on object-oriented verification and record subtyping in HOL.

11.5.1 Basic concepts

Isabelle/HOL supports both *fixed* and *schematic* records at the level of terms and types. The notation is as follows:

	record terms	record types
fixed	$(\!\vert x = a,\ y = b \vert\!)$	$(\!\vert x :: A,\ y :: B \vert\!)$
schematic	$(\!\vert x = a,\ y = b,\ \ldots = m \vert\!)$	$(\!\vert x :: A,\ y :: B,\ \ldots :: M \vert\!)$

The ASCII representation of $(\!\vert x = a \vert\!)$ is $(\vert\ x = a\ \vert)$.
A fixed record $(\!\vert x = a,\ y = b \vert\!)$ has field x of value a and field y of value b. The corresponding type is $(\!\vert x :: A,\ y :: B \vert\!)$, assuming that $a :: A$ and $b :: B$.
A record scheme like $(\!\vert x = a,\ y = b,\ \ldots = m \vert\!)$ contains fields x and y as before, but also possibly further fields as indicated by the "$\ldots$" notation (which is actually part of the syntax). The improper field "$\ldots$" 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 $(\!\vert x = a,\ y = b,\ z = c,\ \ldots = m' \vert\!)$, where m' refers to a different more part. Fixed records are special instances of record schemes, where "$\ldots$" is properly terminated by the $() :: unit$ element. In fact, $(\!\vert x = a,\ y = b \vert\!)$ is just an abbreviation for $(\!\vert x = a,\ y = b,\ \ldots = () \vert\!)$.

Two key observations make extensible records in a simply typed language like HOL work out:

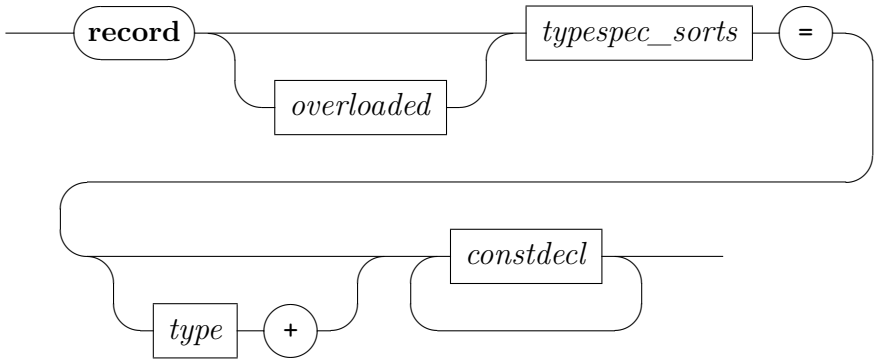
1. the more part is internalized, as a free term or type variable,
2. field names are externalized, they cannot be accessed within the logic as first-class values.

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

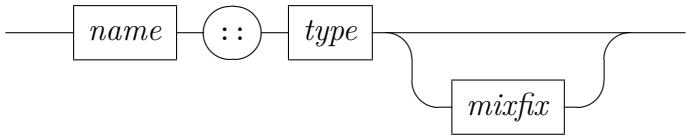
operations like selectors and updates. The common setup for various generic proof tools enable succinct reasoning patterns. See also the Isabelle/HOL tutorial [38] for further instructions on using records in practice.

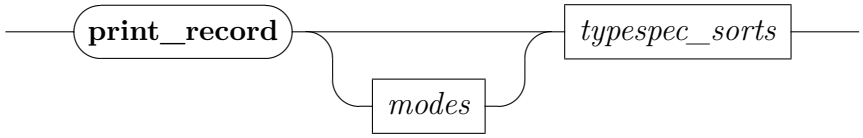
11.5.2 Record specifications

record : *theory* → *theory*
print_record : *context* →

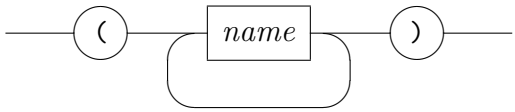


constdecl





modes



record $(\alpha_1, \ldots, \alpha_m)\ t = \tau + c_1 :: \sigma_1 \ \ldots \ c_n :: \sigma_n$ defines extensible record type $(\alpha_1, \ldots, \alpha_m)\ t$, derived from the optional parent record τ by adding new field components $c_i :: \sigma_i$ etc.

The type variables of τ and σ_i need to be covered by the (distinct) parameters $\alpha_1, \ldots, \alpha_m$. Type constructor t has to be new, while τ 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 τ 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_scheme* made an abbreviation for $(\!|c_1 :: \sigma_1, \ldots, c_n :: \sigma_n, \ldots :: \zeta|\!)$.

print_record $(\alpha_1, \ldots, \alpha_m)\ t$ prints the definition of record $(\alpha_1, \ldots, \alpha_m)\ t$. Optionally *modes* can be specified, which are appended to the current print mode; see §8.1.3.

11.5.3 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*"):

$$\begin{array}{lcl} c_i & :: & (\!|\overline{c} :: \overline{\sigma}, \ldots :: \zeta|\!) \Rightarrow \sigma_i \\ c_i_update & :: & (\sigma_i \Rightarrow \sigma_i) \Rightarrow (\!|\overline{c} :: \overline{\sigma}, \ldots :: \zeta|\!) \Rightarrow (\!|\overline{c} :: \overline{\sigma}, \ldots :: \zeta|\!) \end{array}$$

There is special syntax for application of updates: $r(\!|x := a|\!)$ abbreviates term $x_update\ (\lambda_.\ a)\ r$. 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 $(\!|x := a, y := b, z := c|\!)$, 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 **make** operation provides a cumulative record constructor function:

$$t.make \;::\; \sigma_1 \Rightarrow \ldots\ \sigma_n \Rightarrow (\!|\overline{c} :: \overline{\sigma}|\!)$$

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 $b_1 :: \varrho_1, \ldots, b_k :: \varrho_k$, the above record operations will get the following types:

$$\begin{array}{lll} c_i & :: & (\!|\overline{b} :: \overline{\varrho}, \overline{c} :: \overline{\sigma}, \ldots :: \zeta|\!) \Rightarrow \sigma_i \\ c_i_update & :: & (\sigma_i \Rightarrow \sigma_i) \Rightarrow (\!|\overline{b} :: \overline{\varrho}, \overline{c} :: \overline{\sigma}, \ldots :: \zeta|\!) \Rightarrow (\!|\overline{b} :: \overline{\varrho}, \overline{c} :: \overline{\sigma}, \ldots :: \zeta|\!) \\ t.make & :: & \varrho_1 \Rightarrow \ldots\ \varrho_k \Rightarrow \sigma_1 \Rightarrow \ldots\ \sigma_n \Rightarrow (\!|\overline{b} :: \overline{\varrho}, \overline{c} :: \overline{\sigma}|\!) \end{array}$$

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.

$$\begin{array}{lll} t.fields & :: & \sigma_1 \Rightarrow \ldots\ \sigma_n \Rightarrow (\!|\overline{c} :: \overline{\sigma}|\!) \\ t.extend & :: & (\!|\overline{b} :: \overline{\varrho}, \overline{c} :: \overline{\sigma}|\!) \Rightarrow \zeta \Rightarrow (\!|\overline{b} :: \overline{\varrho}, \overline{c} :: \overline{\sigma}, \ldots :: \zeta|\!) \\ t.truncate & :: & (\!|\overline{b} :: \overline{\varrho}, \overline{c} :: \overline{\sigma}, \ldots :: \zeta|\!) \Rightarrow (\!|\overline{b} :: \overline{\varrho}, \overline{c} :: \overline{\sigma}|\!) \end{array}$$

Note that *t.make* and *t.fields* coincide for root records.

11.5.4 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 *simp* method without further arguments. These rules are available as *t.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 $(x, y) = (x', y') \equiv x = x' \wedge y = y'$ are declared to the Simplifier and Classical Reasoner as *iff* rules. These rules are available as *t.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 "*intro?*". The rule is called *t.equality*.

5. Representations of arbitrary record expressions as canonical constructor terms are provided both in *cases* and *induct* format (cf. the generic proof methods of the same name, §6.5). 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 "(*cases r*)" to a certain proof problem.

6. The derived record operations *t.make*, *t.fields*, *t.extend*, *t.truncate* are *not* treated automatically, but usually need to be expanded by hand, using the collective fact *t.defs*.

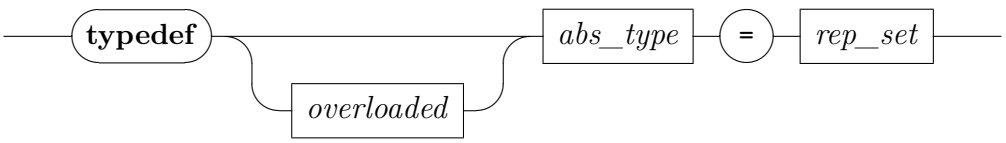
Examples

See `~~/src/HOL/Examples/Records.thy`, for example.

11.6 Semantic subtype definitions

typedef : *local_theory* → *proof*(*prove*)

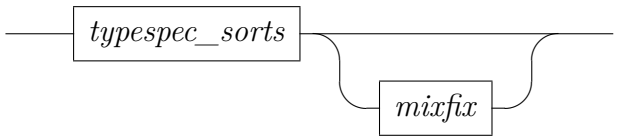
A type definition identifies a new type with a non-empty subset of an existing type. More precisely, the new type is defined by exhibiting an existing type τ, a set $A :: \tau\ set$, and proving $\exists x.\ x \in A$. Thus A is a non-empty subset of τ, and the new type denotes this subset. New functions are postulated that establish an isomorphism between the new type and the subset. In general, the type τ 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.



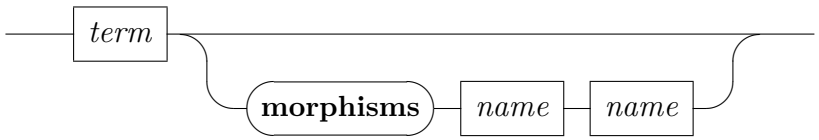
overloaded



abs_type



rep_set



To understand the concept of type definition better, we need to recount its somewhat complex history. The HOL logic goes back to the "Simple Theory of Types" (STT) of A. Church [14], which is further explained in the book by P. Andrews [1]. The overview article by W. Farmer [16] points out the "seven virtues" of this relatively simple family of logics. STT has only ground types, without polymorphism and without type definitions.

M. Gordon [19] augmented Church's STT by adding schematic polymorphism (type variables and type constructors) and a facility to introduce new types as semantic subtypes from existing types. This genuine extension of the logic was explained semantically by A. Pitts in the book of the original Cambridge HOL88 system [50]. Type definitions work in this setting, because the general model-theory of STT is restricted to models that ensure that the universe of type interpretations is closed by forming subsets (via predicates taken from the logic).

Isabelle/HOL goes beyond Gordon-style HOL by admitting overloaded constant definitions [57, 23], which are actually a concept of Isabelle/Pure and do not depend on particular set-theoretic semantics of HOL. Over many years, there was no formal checking of semantic type definitions in Isabelle/HOL versus syntactic constant definitions in Isabelle/Pure. So the **typedef** command was described as "axiomatic" in the sense of §5.5, only with some local checks of the given type and its representing set.

Recent clarification of overloading in the HOL logic proper [28] demonstrates how the dissimilar concepts of constant definitions versus type definitions may be understood uniformly. This requires an interpretation

of Isabelle/HOL that substantially reforms the set-theoretic model of A. Pitts [50], by taking a schematic view on polymorphism and interpreting only ground types in the set-theoretic sense of HOL88. Moreover, type-constructors may be explicitly overloaded, e.g. by making the subset depend on type-class parameters (cf. §5.8). This is semantically like a dependent type: the meaning relies on the operations provided by different type-class instances.

typedef $(\alpha_1, \ldots, \alpha_n)\ t = A$ defines a new type $(\alpha_1, \ldots, \alpha_n)\ t$ from the set A over an existing type. The set A may contain type variables $\alpha_1, \ldots, \alpha_n$ as specified on the LHS, but no term variables. Non-emptiness of A needs to be proven on the spot, in order to turn the internal conditional characterization into usable theorems.

The "(*overloaded*)" option allows the **typedef** specification to depend on constants that are not (yet) specified and thus left open as parameters, e.g. type-class parameters.

Within a local theory specification, the newly introduced type constructor cannot depend on parameters or assumptions of the context: this is syntactically impossible in HOL. The non-emptiness proof may formally depend on local assumptions, but this has little practical relevance.

For **typedef** $t = A$ the newly introduced type t is accompanied by a pair of morphisms to relate it to the representing set over the old type. By default, the injection from type to set is called *Rep_t* and its inverse *Abs_t*: An explicit **morphisms** specification allows to provide alternative names.

The logical characterization of **typedef** uses the predicate of locale *type_definition* that is 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 theorem *type_definition.Rep* is turned into the specific *Rep_t*, for example.

Theorems *type_definition.Rep*, *type_definition.Rep_inverse*, and *type_definition.Abs_inverse* provide the most basic characterization as a corresponding injection/surjection pair (in both directions). The derived rules *type_definition.Rep_inject* and *type_definition.Abs_inject* provide a more convenient version of injectivity, suitable for automated proof tools (e.g. in declarations involving *simp* or *iff*). Furthermore, the rules *type_definition.Rep_cases* / *type_definition.Rep_induct*, and

type_definition.Abs_cases / *type_definition.Abs_induct* provide alternative views on surjectivity. These rules are already declared as set or type rules for the generic *cases* and *induct* methods, respectively.

Examples

The following trivial example pulls a three-element type into existence within the formal logical environment of Isabelle/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* ≠ *Two* *One* ≠ *Three* *Two* ≠ *Three*
  by (*simp_all add*: *One_def Two_def Three_def Abs_three_inject*)

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*)

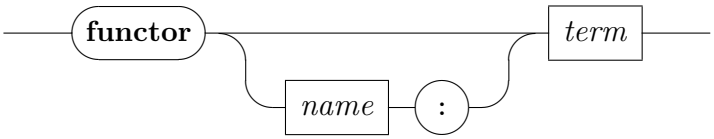
Note that such trivial constructions are better done with derived specification mechanisms such as **datatype**:

datatype *three* = *One* | *Two* | *Three*

This avoids re-doing basic definitions and proofs from the primitive **typedef** above.

11.7 Functorial structure of types

functor : *local_theory* → *proof*(*prove*)



functor *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:

$$m \;::\; \sigma_1 \Rightarrow \ldots\ \sigma_k \Rightarrow (\overline{\alpha}_n)\ t \Rightarrow (\overline{\beta}_n)\ t$$

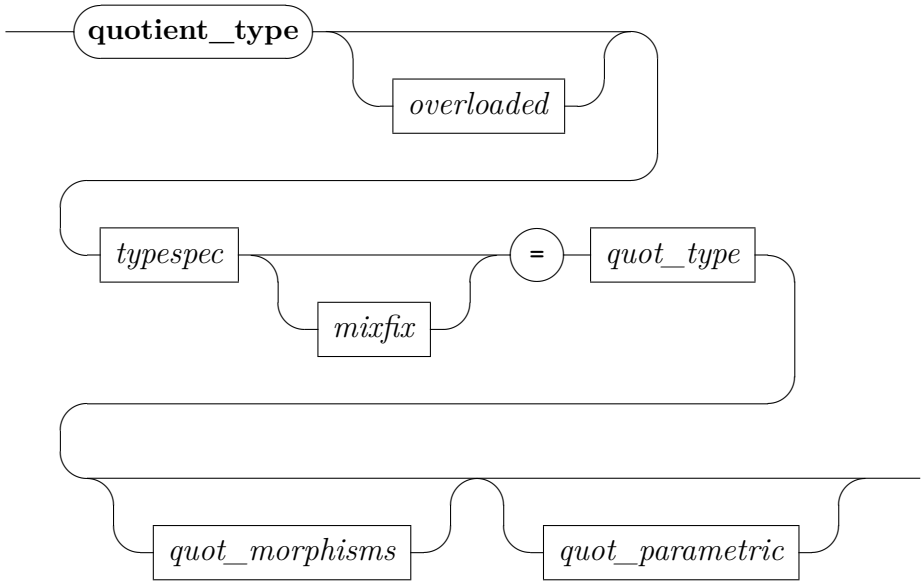
where t is the type constructor, $\overline{\alpha}_n$ and $\overline{\beta}_n$ are distinct type variables free in the local theory and $\sigma_1, \ldots, \sigma_k$ is a subsequence of $\alpha_1 \Rightarrow \beta_1, \beta_1 \Rightarrow \alpha_1, \ldots, \alpha_n \Rightarrow \beta_n, \beta_n \Rightarrow \alpha_n$.

11.8 Quotient types with lifting and transfer

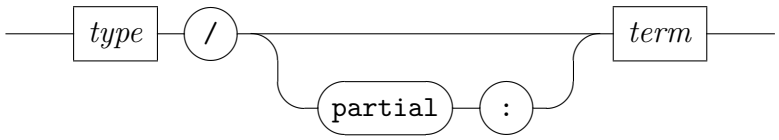
The quotient package defines a new quotient type given a raw type and a partial equivalence relation (§11.8.1). The package also historically includes automation for transporting definitions and theorems (§11.8.4), but most of this automation was superseded by the Lifting (§11.8.2) and Transfer (§11.8.3) packages.

11.8.1 Quotient type definition

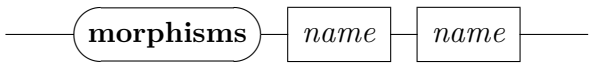
quotient_type : *local_theory* → *proof*(*prove*)



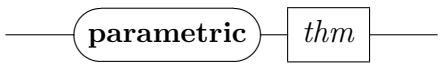
quot_type



quot_morphisms



quot_parametric



quotient_type defines a new quotient type τ. The injection from a quotient type to a raw type is called *rep_*τ, its inverse *abs_*τ unless explicit **morphisms** specification provides alternative names. **quotient_type** requires the user to prove that the relation is an equivalence relation (predicate *equivp*), unless the user specifies explicitly *partial* in which case the obligation is *part_equivp*. A quotient defined with *partial* is weaker in the sense that less things can be proved automatically.

The command internally proves a Quotient theorem and sets up the Lifting package by the command **setup_lifting**. Thus the Lifting and Transfer packages can be used also with quotient types defined by

quotient_type without any extra set-up. The parametricity theorem for the equivalence relation R can be provided as an extra argument of the command and is passed to the corresponding internal call of **setup_lifting**. This theorem allows the Lifting package to generate a stronger transfer rule for equality.

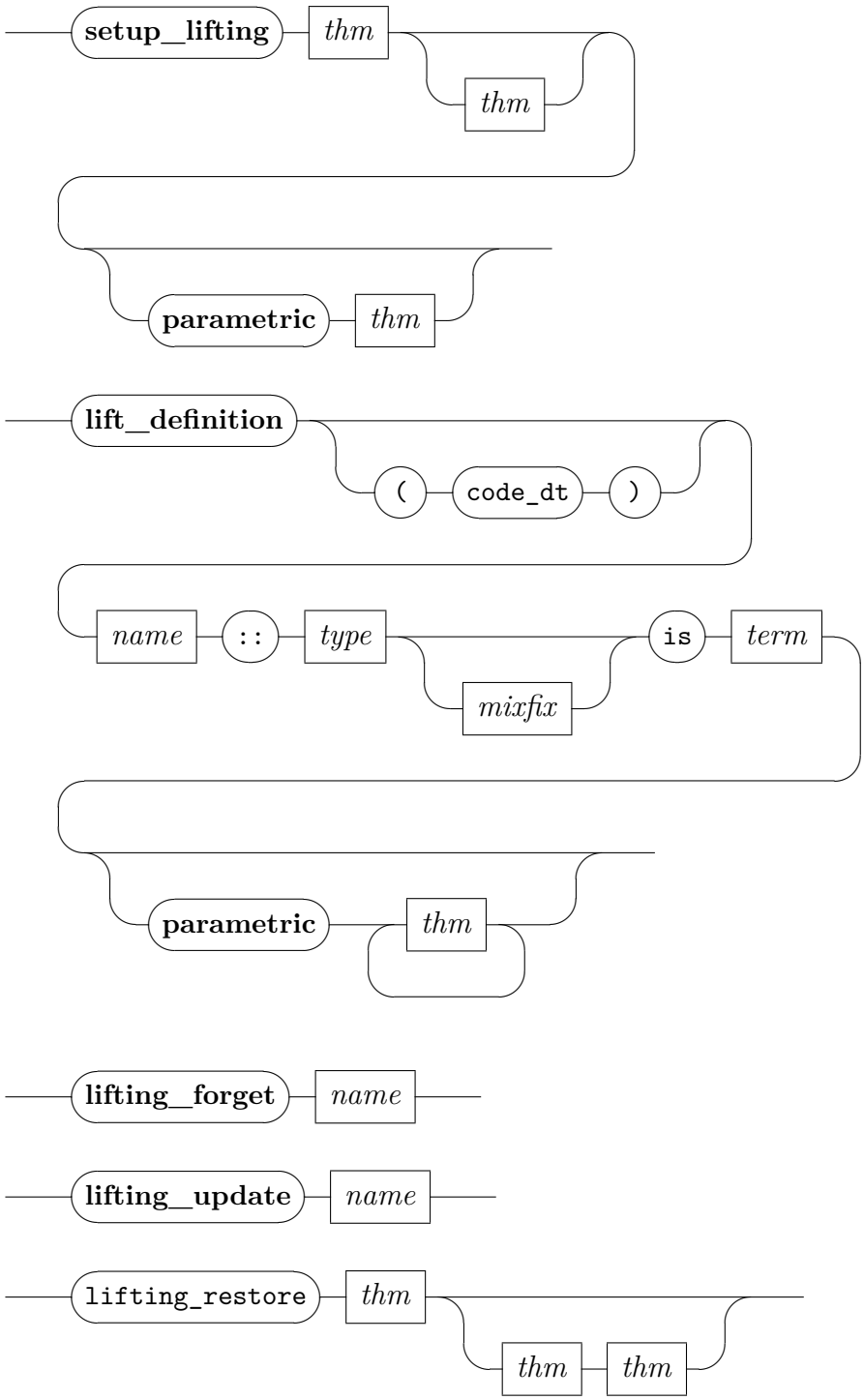
11.8.2 Lifting package

The Lifting package allows users to lift terms of the raw type to the abstract type, which is a necessary step in building a library for an abstract type. Lifting defines a new constant by combining coercion functions (*Abs* and *Rep*) with the raw term. It also proves an appropriate transfer rule for the Transfer (§11.8.3) package and, if possible, an equation for the code generator.

The Lifting package provides two main commands: **setup_lifting** for initializing the package to work with a new type, and **lift_definition** for lifting constants. The Lifting package works with all four kinds of type abstraction: type copies, subtypes, total quotients and partial quotients.

Theoretical background can be found in [24].

setup_lifting	:	*local_theory* → *local_theory*
lift_definition	:	*local_theory* → *proof*(*prove*)
lifting_forget	:	*local_theory* → *local_theory*
lifting_update	:	*local_theory* → *local_theory*
print_quot_maps	:	*context* →
print_quotients	:	*context* →
quot_map	:	*attribute*
relator_eq_onp	:	*attribute*
relator_mono	:	*attribute*
relator_distr	:	*attribute*
quot_del	:	*attribute*
lifting_restore	:	*attribute*



setup_lifting Sets up the Lifting package to work with a user-defined type. The command supports two modes.

1. The first one is a low-level mode when the user must provide as a first argument of **setup_lifting** a quotient theorem *Quotient R Abs Rep T*. The package configures a transfer rule for equality, a domain transfer rules and sets up the **lift_definition** command to work with the abstract type. An optional theorem *reflp R*, which certifies that the equivalence relation R is total, can be provided as a second argument. This allows the package to generate stronger transfer rules. And finally, the parametricity theorem for *R* can be provided as a third argument. This allows the package to generate a stronger transfer rule for equality.

   Users generally will not prove the *Quotient* theorem manually for new types, as special commands exist to automate the process.

2. When a new subtype is defined by **typedef**, **lift_definition** can be used in its second mode, where only the *type_definition* theorem *type_definition Rep Abs A* is used as an argument of the command. The command internally proves the corresponding *Quotient* theorem and registers it with **setup_lifting** using its first mode.

For quotients, the command **quotient_type** can be used. The command defines a new quotient type and similarly to the previous case, the corresponding Quotient theorem is proved and registered by **setup_lifting**.

The command **setup_lifting** also sets up the code generator for the new type. Later on, when a new constant is defined by **lift_definition**, the Lifting package proves and registers a code equation (if there is one) for the new constant.

lift_definition $f :: \tau$ **is** t Defines a new function f with an abstract type τ in terms of a corresponding operation t on a representation type. More formally, if $t :: \sigma$, then the command builds a term F as a corresponding combination of abstraction and representation functions such that $F :: \sigma \Rightarrow \tau$ and defines $f \equiv F\ t$. The term t does not have to be necessarily a constant but it can be any term.

The command opens a proof and the user must discharge a respectfulness proof obligation. For a type copy, i.e. a typedef with *UNIV*, the obligation is discharged automatically. The proof goal 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.

The user can specify a parametricity theorems for t after the keyword **parametric**, which allows the command to generate parametric transfer rules for *f*.

For each constant defined through trivial quotients (type copies or subtypes) *f.rep_eq* is generated. The equation is a code certificate that defines f using the representation function.

For each constant *f.abs_eq* is generated. The equation is unconditional for total quotients. The equation defines f using the abstraction function.

Integration with [*code* abstract]: For subtypes (e.g. corresponding to a datatype invariant, such as *'a dlist*), **lift_definition** uses a code certificate theorem *f.rep_eq* as a code equation. Because of the limitation of the code generator, *f.rep_eq* cannot be used as a code equation if the subtype occurs inside the result type rather than at the top level (e.g. function returning *'a dlist option* vs. *'a dlist*).

In this case, an extension of **lift_definition** can be invoked by specifying the flag *code_dt*. This extension enables code execution through series of internal type and lifting definitions if the return type τ meets the following inductive conditions:

- τ is a type variable
- $\tau = \tau_1 \ldots \tau_n \kappa$, where κ is an abstract type constructor and $\tau_1 \ldots \tau_n$ do not contain abstract types (i.e. *int dlist* is allowed whereas *int dlist dlist* not)
- $\tau = \tau_1 \ldots \tau_n \kappa$, κ is a type constructor that was defined as a (co)datatype whose constructor argument types do not contain either non-free datatypes or the function type.

Integration with [*code* equation]: For total quotients, **lift_definition** uses *f.abs_eq* as a code equation.

lifting_forget and **lifting_update** These two commands serve for storing and deleting the set-up of the Lifting package and corresponding transfer rules defined by this package. This is useful for hiding of type construction details of an abstract type when the construction is finished but it still allows additions to this construction when this is later necessary.

Whenever the Lifting package is set up with a new abstract type τ by **lift_definition**, the package defines a new bundle that is called

τ.*lifting*. This bundle already includes set-up for the Lifting package. The new transfer rules introduced by **lift_definition** can be stored in the bundle by the command **lifting_update** τ.*lifting*.

The command **lifting_forget** τ.*lifting* deletes set-up of the Lifting package for τ and deletes all the transfer rules that were introduced by **lift_definition** using τ as an abstract type.

The stored set-up in a bundle can be reintroduced by the Isar commands for including a bundle (**include**, **includes** and **including**).

print_quot_maps prints stored quotient map theorems.

print_quotients prints stored quotient theorems.

quot_map registers a quotient map theorem, a theorem showing how to "lift" quotients over type constructors. E.g. *Quotient R Abs Rep T* $\Longrightarrow$ *Quotient* (*rel_set R*) (*image Abs*) (*image Rep*) (*rel_set T*). For examples see `~~/src/HOL/Lifting_Set.thy` or `~~/src/HOL/Lifting.thy`. This property is proved automatically if the involved type is BNF without dead variables.

relator_eq_onp registers a theorem that shows that a relator applied to an equality restricted by a predicate P (i.e. *eq_onp P*) is equal to a predicator applied to the P. The combinator *eq_onp* is used for internal encoding of proper subtypes. Such theorems allows the package to hide *eq_onp* from a user in a user-readable form of a respectfulness theorem. For examples see `~~/src/HOL/Lifting_Set.thy` or `~~/src/HOL/Lifting.thy`. This property is proved automatically if the involved type is BNF without dead variables.

relator_mono registers a property describing a monotonicity of a relator. E.g. $A \leq B \Longrightarrow$ *rel_set A* $\leq$ *rel_set B*. This property is needed for proving a stronger transfer rule in **lift_definition** when a parametricity theorem for the raw term is specified and also for the reflexivity prover. For examples see `~~/src/HOL/Lifting_Set.thy` or `~~/src/HOL/Lifting.thy`. This property is proved automatically if the involved type is BNF without dead variables.

relator_distr registers a property describing a distributivity of the relation composition and a relator. E.g. *rel_set R* $\circ\circ$ *rel_set S* = *rel_set* (R $\circ\circ$ S). This property is needed for proving a stronger transfer rule in **lift_definition** when a parametricity theorem for the raw term is specified. When this equality does not hold unconditionally

(e.g. for the function type), the user can specified each direction separately and also register multiple theorems with different set of assumptions. This attribute can be used only after the monotonicity property was already registered by *relator_mono*. For examples see `~~/src/HOL/Lifting_Set.thy` or `~~/src/HOL/Lifting.thy`. This property is proved automatically if the involved type is BNF without dead variables.

quot_del deletes a corresponding Quotient theorem from the Lifting infrastructure and thus de-register the corresponding quotient. This effectively causes that **lift_definition** will not do any lifting for the corresponding type. This attribute is rather used for low-level manipulation with set-up of the Lifting package because **lifting_forget** is preferred for normal usage.

lifting_restore Quotient_thm pcr_def pcr_cr_eq_thm registers the Quotient theorem *Quotient_thm* in the Lifting infrastructure and thus sets up lifting for an abstract type τ (that is defined by *Quotient_thm*). Optional theorems *pcr_def* and *pcr_cr_eq_thm* can be specified to register the parametrized correspondence relation for τ. E.g. for *'a dlist*, *pcr_def* is $pcr_dlist\ A \equiv list_all2\ A \circ\circ cr_dlist$ and *pcr_cr_eq_thm* is $pcr_dlist\ (=) = (=)$. This attribute is rather used for low-level manipulation with set-up of the Lifting package because using of the bundle τ*.lifting* together with the commands **lifting_forget** and **lifting_update** is preferred for normal usage.

Integration with the BNF package [8]: As already mentioned, the theorems that are registered by the following attributes are proved and registered automatically if the involved type is BNF without dead variables: *quot_map*, *relator_eq_onp*, *relator_mono*, *relator_distr*. Also the definition of a relator and predicator is provided automatically. Moreover, if the BNF represents a datatype, simplification rules for a predicator are again proved automatically.

11.8.3 Transfer package

transfer	:	*method*
transfer′	:	*method*
transfer_prover	:	*method*
Transfer.transferred	:	*attribute*
untransferred	:	*attribute*
transfer_start	:	*method*
transfer_prover_start	:	*method*
transfer_step	:	*method*
transfer_end	:	*method*
transfer_prover_end	:	*method*
transfer_rule	:	*attribute*
transfer_domain_rule	:	*attribute*
relator_eq	:	*attribute*
relator_domain	:	*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_start, *transfer_step*, *transfer_end*, *transfer_prover_start* and *transfer_prover_end* methods are meant to be used for debugging of *transfer* and *transfer_prover*, which we can decompose as follows: *transfer* = (*transfer_start*, *transfer_step*+, *transfer_end*) and *transfer_prover* = (*transfer_prover_start*, *transfer_step*+, *transfer_prover_end*). For usage examples see `~~/src/HOL/ex/Transfer_Debug.thy`.

untransferred proves the same equivalent theorem as *transfer* internally does.

Transfer.transferred works in the opposite direction than *transfer'*. E.g. given the transfer relation $ZN\ x\ n \equiv (x = int\ n)$, corresponding transfer rules and the theorem $\forall\, x{::}int \in \{0..\}.\ x < x + 1$, the attribute would prove $\forall\, n{::}nat.\ n < n + 1$. The attribute is still in experimental phase of development.

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:

$((A \mathrel{===>} B) \mathrel{===>} list_all2\ A \mathrel{===>} list_all2\ B)\ map\ map$
$(cr_int \mathrel{===>} cr_int \mathrel{===>} cr_int)\ (\lambda(x,y)\ (u,v).\ (x+u,\ y+v))\ plus$

Lemmas involving predicates on relations can also be registered using the same attribute. For example:

$bi_unique\ A \Longrightarrow (list_all2\ A \mathrel{===>} (=))\ distinct\ distinct$
$[\![bi_unique\ A;\ bi_unique\ B]\!] \Longrightarrow bi_unique\ (rel_prod\ A\ B)$

Preservation of predicates on relations (*bi_unique*, *bi_total*, *right_unique*, *right_total*, *left_unique*, *left_total*) with the respect to a relator is proved automatically if the involved type is BNF [8] without dead variables.

transfer_domain_rule attribute maintains a collection of rules, which specify a domain of a transfer relation by a predicate. E.g. given the transfer relation $ZN\ x\ n \equiv (x = int\ n)$, one can register the following transfer domain rule: $Domainp\ ZN = (\lambda x.\ x \geq 0)$. The rules allow the package to produce more readable transferred goals, e.g. when quantifiers are transferred.

relator_eq attribute collects identity laws for relators of various type constructors, e.g. $rel_set\ (=) = (=)$. The *transfer* method uses these lemmas to infer transfer rules for non-polymorphic constants on the fly. For examples see `~~/src/HOL/Lifting_Set.thy` or `~~/src/HOL/Lifting.thy`. This property is proved automatically if the involved type is BNF without dead variables.

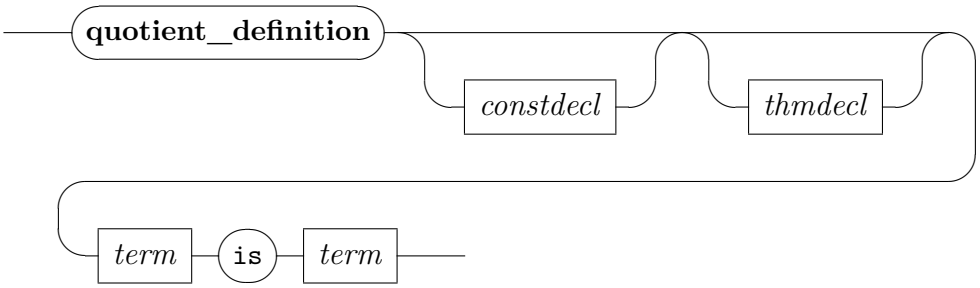
relator_domain attribute collects rules describing domains of relators by predicators. E.g. $Domainp\ (rel_set\ T) = (\lambda A.\ Ball\ A\ (Domainp\ T))$.

This allows the package to lift transfer domain rules through type constructors. For examples see `~~/src/HOL/Lifting_Set.thy` or `~~/src/HOL/Lifting.thy`. This property is proved automatically if the involved type is BNF without dead variables.

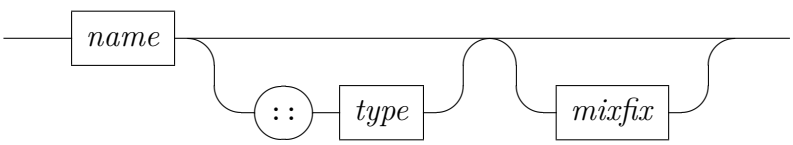
Theoretical background can be found in [24].

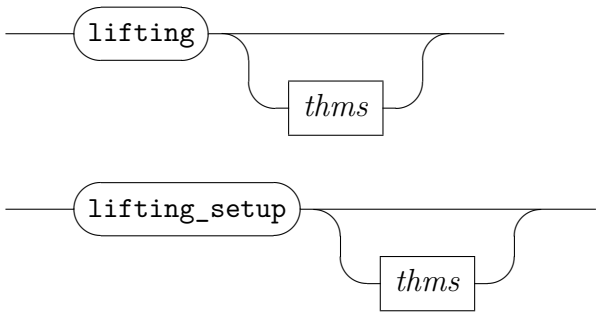
11.8.4 Old-style definitions for quotient types

quotient_definition	:	*local_theory* → *proof*(*prove*)
print_quotmapsQ3	:	*context* →
print_quotientsQ3	:	*context* →
print_quotconsts	:	*context* →
lifting	:	*method*
lifting_setup	:	*method*
descending	:	*method*
descending_setup	:	*method*
partiality_descending	:	*method*
partiality_descending_setup	:	*method*
regularize	:	*method*
injection	:	*method*
cleaning	:	*method*
quot_thm	:	*attribute*
quot_lifted	:	*attribute*
quot_respect	:	*attribute*
quot_preserve	:	*attribute*



constdecl





quotient_definition defines a constant on the quotient type.

print_quotmapsQ3 prints quotient map functions.

print_quotientsQ3 prints quotients.

print_quotconsts prints quotient constants.

lifting and *lifting_setup* methods match the current goal with the given raw theorem to be lifted producing three new subgoals: regularization, injection and cleaning subgoals. *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 *lifting_setup* which leaves the three subgoals unsolved.

descending and *descending_setup* try to guess a raw statement that would lift to the current subgoal. Such statement is assumed as a new subgoal and *descending* continues in the same way as *lifting* does. *descending* tries to solve the arising regularization, injection and cleaning subgoals with the analogous method *descending_setup* which leaves the four unsolved subgoals.

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. *partiality_descending_setup* leaves the injection and cleaning subgoals unchanged.

regularize applies the regularization heuristics to the current subgoal.

injection applies the injection heuristics to the current goal using the stored quotient respectfulness theorems.

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 **functor** and a relation map defined for for the container type, the quotient extension theorem should be $Quotient3\ R\ Abs\ Rep \Longrightarrow Quotient3\ (rel_map\ R)\ (map\ Abs)\ (map\ Rep)$. Quotient extension theorems are stored in a database and are used all the steps of lifting theorems.

Chapter 12

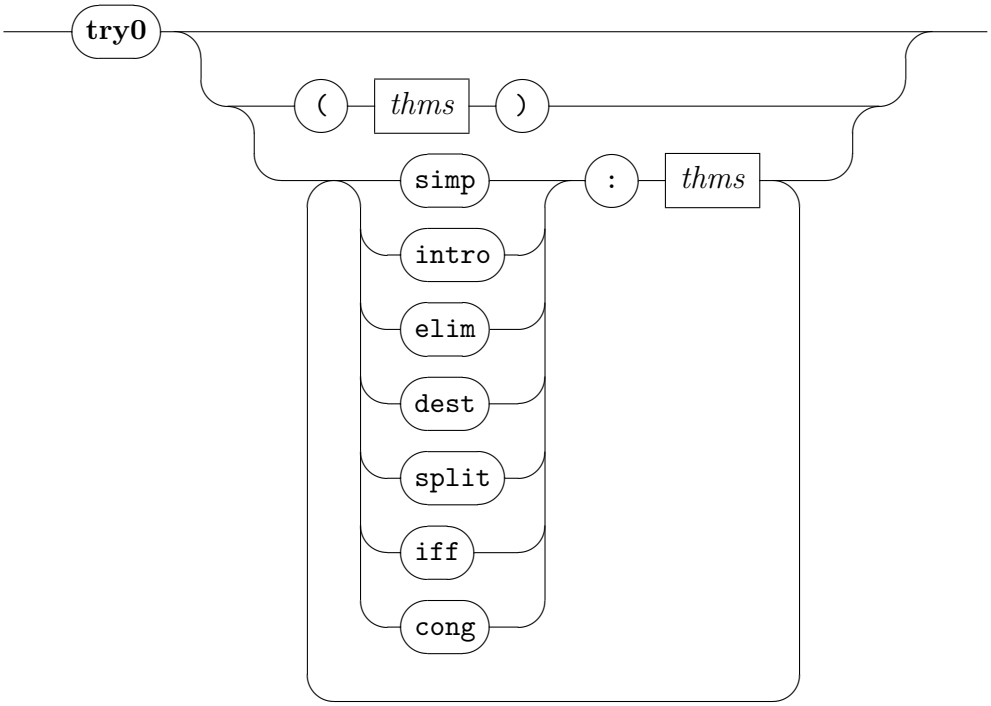
Proof tools

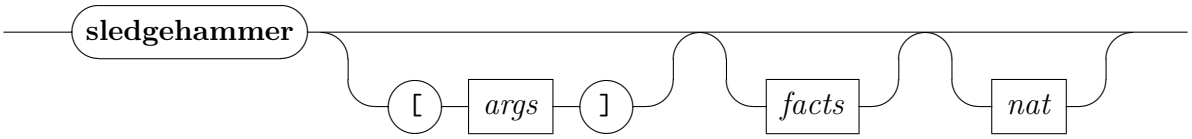
12.1 Proving propositions

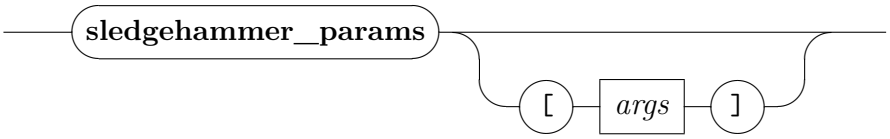
In addition to the standard proof methods, a number of diagnosis tools search for proofs and provide an Isar proof snippet on success. These tools are available via the following commands.

solve_direct*	:	*proof* $\rightarrow$
try*	:	*proof* $\rightarrow$
try0*	:	*proof* $\rightarrow$
sledgehammer*	:	*proof* $\rightarrow$
sledgehammer_params	:	*theory* $\rightarrow$ *theory*

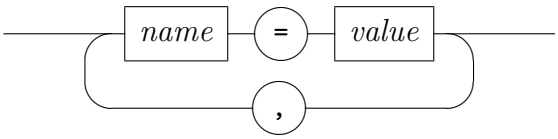



try0
(
thms
)
simp
:
thms
intro
elim
dest
split
iff
cong

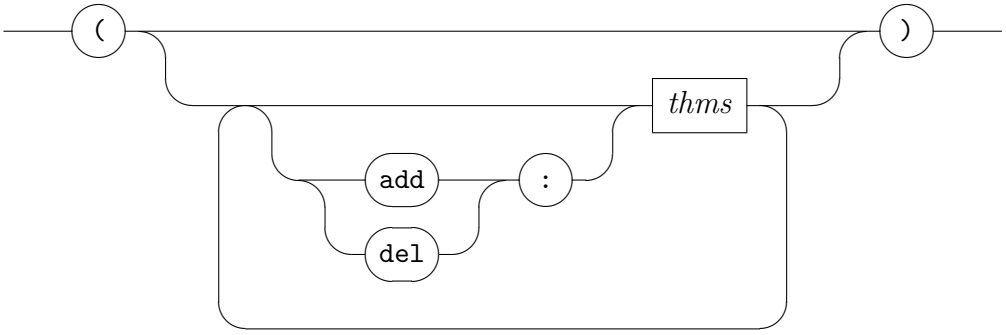

sledgehammer
[
args
]
facts
nat


sledgehammer_params
[
args
]

args


name
=
value
,

facts



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*:, *dest*:, *split*:, *iff*:, and *cong*: are passed to the appropriate proof methods. Proof methods that do not accept a kind of fact receive it via **using**. Alternatively, facts in parentheses are passed without a kind; the fact classifier selected by the configuration option *try*0*_fact_classifier* may assign kinds to them.

try attempts to prove or disprove a subgoal using a combination of provers and disprovers (**solve_direct**, **quickcheck**, **try0**, **sledgehammer**, **nitpick**).

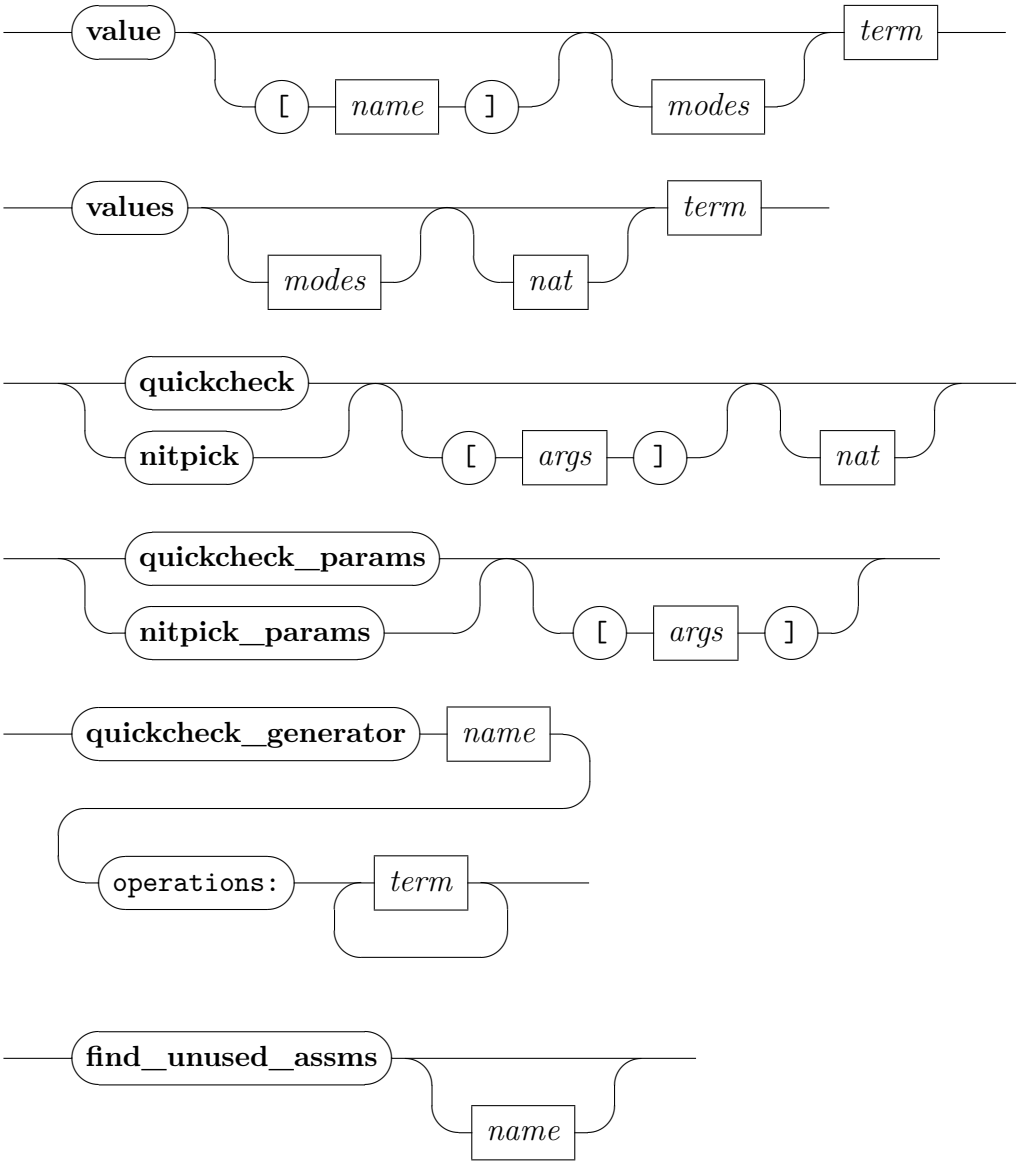
sledgehammer attempts to prove a subgoal using external automatic provers (resolution provers and SMT solvers). See the Sledgehammer manual [9] for details.

sledgehammer_params changes **sledgehammer** configuration options persistently.

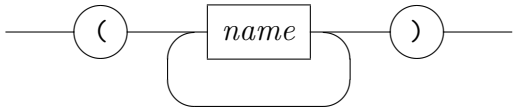
12.2 Checking and refuting propositions

Identifying incorrect propositions usually involves evaluation of particular assignments and systematic counterexample search. This is supported by the following commands.

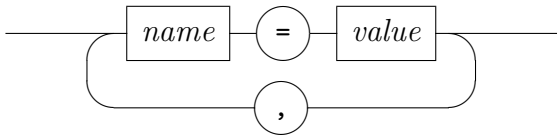
value* : *context* $\rightarrow$
values* : *context* $\rightarrow$
quickcheck* : *proof* $\rightarrow$
nitpick* : *proof* $\rightarrow$
quickcheck_params : *theory* $\rightarrow$ *theory*
nitpick_params : *theory* $\rightarrow$ *theory*
quickcheck_generator : *theory* $\rightarrow$ *theory*
find_unused_assms : *context* $\rightarrow$



modes



args



value *t* evaluates and prints a term; optionally *modes* can be specified, which are appended to the current print mode; see §8.1.3. Evaluation is tried first using ML, falling back to normalization by evaluation if this fails. 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 §8.1.3.

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. This option is currently only supported by the default (exhaustive) tester.

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.

Using the following type classes, the testers generate values and convert them back into Isabelle terms for displaying counterexamples.

exhaustive The parameters of the type classes *exhaustive* and *full_exhaustive* implement the testing. They take a testing function as a parameter, which takes a value of type *'a* and optionally produces a counterexample, and a size parameter for the test values. In *full_exhaustive*, the testing function parameter additionally expects a lazy term reconstruction in the type *Code_Evaluation.term* of the tested value.

The canonical implementation for *exhaustive* testers calls the given testing function on all values up to the given size and stops as soon as a counterexample is found.

random The operation *Quickcheck_Random.random* of the type class *random* generates a pseudo-random value of the given size and a lazy term reconstruction of the value in the type *Code_Evaluation.term*. A pseudo-randomness generator is defined in theory *HOL.Random*.

narrowing implements Haskell's Lazy Smallcheck [51] using the type classes *narrowing* and *partial_term_of*. Variables in the current goal are initially represented as symbolic variables. If the execution of the goal tries to evaluate one of them, the test engine replaces it with refinements provided by *narrowing*. Narrowing views every value as a sum-of-products which is expressed using the operations *Quickcheck_Narrowing.cons* (embedding a value), *Quickcheck_Narrowing.apply* (product) and *Quickcheck_Narrowing.sum* (sum). The refinement should enable further evaluation of the goal.

For example, *narrowing* for the list type *'a* :: *narrowing list* can be recursively defined as *Quickcheck_Narrowing.sum* (*Quickcheck_Narrowing.cons* []) (*Quickcheck_Narrowing.apply* (*Quickcheck_Narrowing.apply* (*Quickcheck_Narrowing.cons* (#)) *narrowing*) *narrowing*). If a symbolic variable of type _ *list* is evaluated, it is replaced by (i) the empty list [] and (ii) by a non-empty list whose head and tail can then be recursively refined if needed.

To reconstruct counterexamples, the operation *partial_term_of* transforms *narrowing*'s deep representation of terms to the type *Code_Evaluation.term*. The deep representation models symbolic variables as *Quickcheck_Narrowing.Narrowing_variable*, which

are normally converted to *Code_Evaluation.Free*, and refined values as *Quickcheck_Narrowing.Narrowing_constructor i args*, where *i* :: *integer* denotes the index in the sum of refinements. In the above example for lists, 0 corresponds to [] and 1 to (#).

The command **code_datatype** sets up *partial_term_of* such that the *i*-th refinement is interpreted as the *i*-th constructor, but it does not ensures consistency with *narrowing*.

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.

nitpick tests the current goal for counterexamples using a reduction to first-order relational logic. See the Nitpick manual [10] for details.

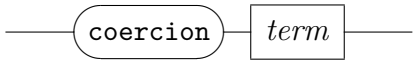
nitpick_params changes **nitpick** configuration options persistently.

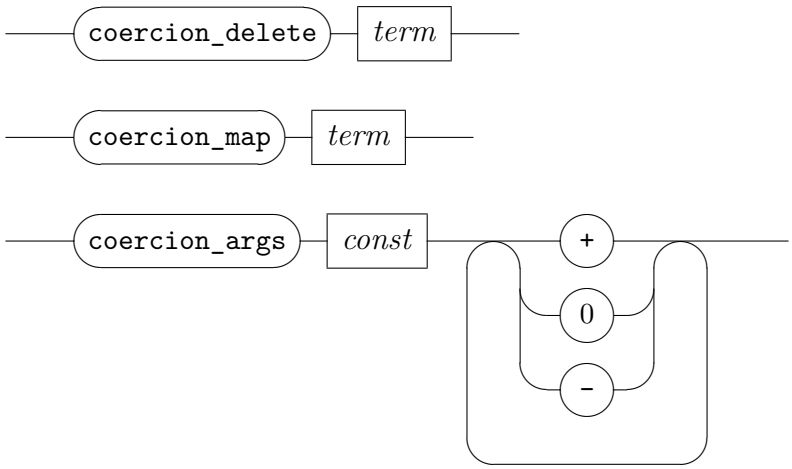
find_unused_assms finds potentially superfluous assumptions in theorems using quickcheck. It takes the theory name to be checked for superfluous assumptions as optional argument. If not provided, it checks the current theory. Options to the internal quickcheck invocations can be changed with common configuration declarations.

12.3 Coercive subtyping

coercion : *attribute*
coercion_delete : *attribute*
coercion_enabled : *attribute*
coercion_map : *attribute*
coercion_args : *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 [53] for details.





coercion f registers a new coercion function $f :: \sigma_1 \Rightarrow \sigma_2$ where σ_1 and σ_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_delete f deletes a preceding declaration (using *coercion*) of the function $f :: \sigma_1 \Rightarrow \sigma_2$ as a coercion.

coercion_map map registers a new map function to lift coercions through type constructors. The function *map* must conform to the following type pattern

$$map \;::\; f_1 \Rightarrow \ldots \Rightarrow f_n \Rightarrow (\alpha_1, \ldots, \alpha_n)\ t \Rightarrow (\beta_1, \ldots, \beta_n)\ t$$

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_args can be used to disallow coercions to be inserted in certain positions in a term. For example, given the constant $c :: \sigma_1 \Rightarrow \sigma_2 \Rightarrow \sigma_3 \Rightarrow \sigma_4$ and the list of policies $-\ +\ 0$ as arguments, coercions will not be inserted in the first argument of c (policy $-$); they may be inserted in the second argument (policy $+$) even if the constant c itself is in a position where coercions are disallowed; the third argument inherits the allowance of coercsion insertion from the position of the constant c (policy 0). The standard usage of policies is the definition of syntatic

constructs (usually extralogical, i.e., processed and stripped during type inference), that should not be destroyed by the insertion of coercions (see, for example, the setup for the case syntax in *HOL.Ctr_Sugar*).

coercion_enabled enables the coercion inference algorithm.

12.4 Arithmetic proof support

arith : *method*
arith : *attribute*
linarith_split : *attribute*

arith decides linear arithmetic problems (on types *nat*, *int*, *real*). Any current facts are inserted into the goal before running the procedure.

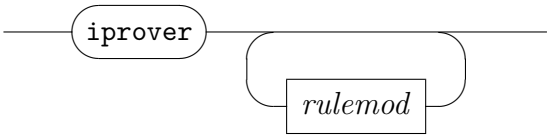
arith declares facts that are supplied to the arithmetic provers implicitly.

linarith_split attribute declares case split rules to be expanded before *arith* is invoked.

Note that a simpler (but faster) arithmetic prover is already invoked by the Simplifier.

12.5 Intuitionistic proof search

iprover : *method*

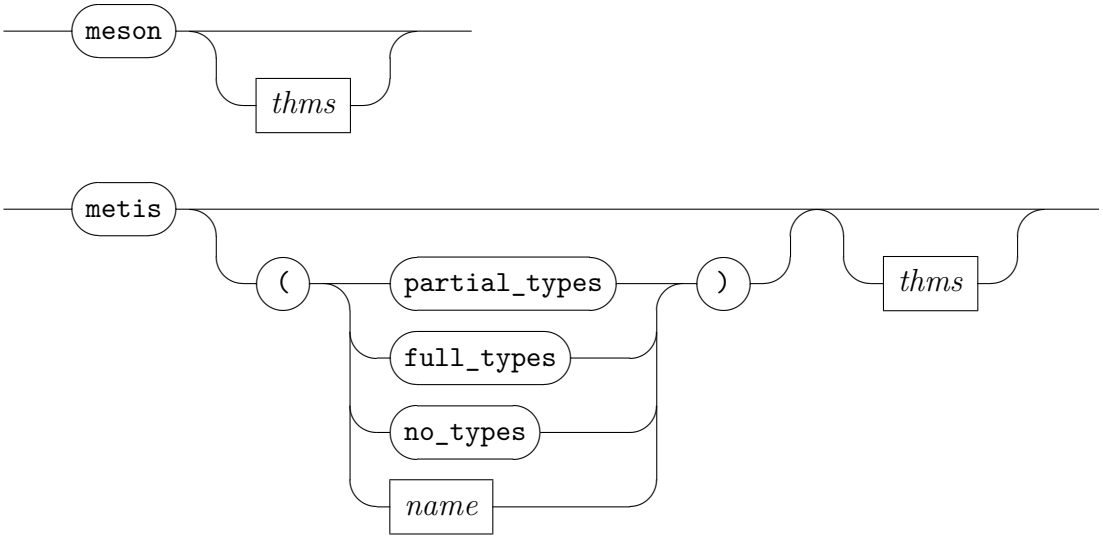


iprover performs intuitionistic proof search, depending on specifically declared rules from the context, or given as explicit arguments. Chained facts are inserted into the goal before commencing proof search.

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.

12.6 Model Elimination and Resolution

meson : *method*
metis : *method*

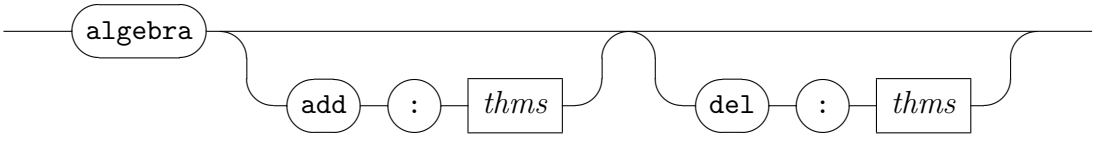


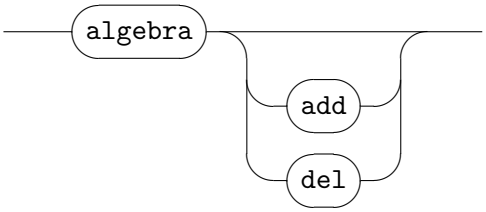
meson implements Loveland's model elimination procedure [30]. See `~~/src/HOL/ex/Meson_Test.thy` for examples.

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 [9] for details. The directory `~~/src/HOL/Metis_Examples` contains several small theories developed to a large extent using *metis*.

12.7 Algebraic reasoning via Gröbner bases

algebra : *method*
algebra : *attribute*





algebra performs algebraic reasoning via Gröbner bases, see also [13] and [12, §3.2]. The method handles deals with two main classes of problems:

1. Universal problems over multivariate polynomials in a (semi)-ring/field/idom; the capabilities of the method are augmented according to properties of these structures. For this problem class the method is only complete for algebraically closed fields, since the underlying method is based on Hilbert's Nullstellensatz, where the equivalence only holds for algebraically closed fields.

   The problems can contain equations $p = 0$ or inequations $q \neq 0$ anywhere within a universal problem statement.

2. All-exists problems of the following restricted (but useful) form:

   $$\begin{array}{l}\forall\, x_1\ \ldots\ x_n.\\ \quad e_1(x_1,\ \ldots,\ x_n) = 0 \wedge \ \ldots\ \wedge\ e_m(x_1,\ \ldots,\ x_n) = 0 \longrightarrow\\ \quad (\exists\, y_1\ \ldots\ y_k.\\ \quad\ p_{11}(x_1,\ \ldots\ ,x_n) * y_1 + \ \ldots\ + p_{1k}(x_1,\ \ldots,\ x_n) * y_k = 0 \wedge\\ \quad\ \ldots\ \wedge\\ \quad\ p_{t1}(x_1,\ \ldots,\ x_n) * y_1 + \ \ldots\ + p_{tk}(x_1,\ \ldots,\ x_n) * y_k = 0)\end{array}$$

   Here $e_1,\ \ldots,\ e_n$ and the p_{ij} are multivariate polynomials only in the variables mentioned as arguments.

The proof method is preceded by a simplification step, which may be modified by using the form (*algebra add*: ths_1 *del*: ths_2). This acts like declarations for the Simplifier (§9.3) on a private simpset for this tool.

algebra (as attribute) manages the default collection of pre-simplification rules of the above proof method.

Example

The subsequent example is from geometry: collinearity is invariant by rotation.

type_synonym *point* = *int* × *int*

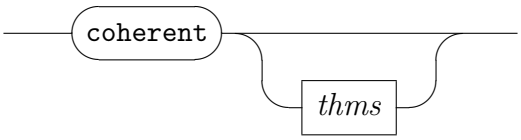
fun *collinear* :: *point* ⇒ *point* ⇒ *point* ⇒ *bool* **where**
collinear (*Ax*, *Ay*) (*Bx*, *By*) (*Cx*, *Cy*) ⟷
(*Ax* − *Bx*) ∗ (*By* − *Cy*) = (*Ay* − *By*) ∗ (*Bx* − *Cx*)

lemma *collinear_inv_rotation*:
assumes *collinear* (*Ax*, *Ay*) (*Bx*, *By*) (*Cx*, *Cy*) **and** $c^2 + s^2 = 1$
shows *collinear* (*Ax* ∗ *c* − *Ay* ∗ *s*, *Ay* ∗ *c* + *Ax* ∗ *s*)
(*Bx* ∗ *c* − *By* ∗ *s*, *By* ∗ *c* + *Bx* ∗ *s*) (*Cx* ∗ *c* − *Cy* ∗ *s*, *Cy* ∗ *c* + *Cx* ∗ *s*)
using *assms* **by** (*algebra add*: *collinear.simps*)

See also `~~/src/HOL/Examples/Groebner_Examples.thy`.

12.8 Coherent Logic

coherent : *method*

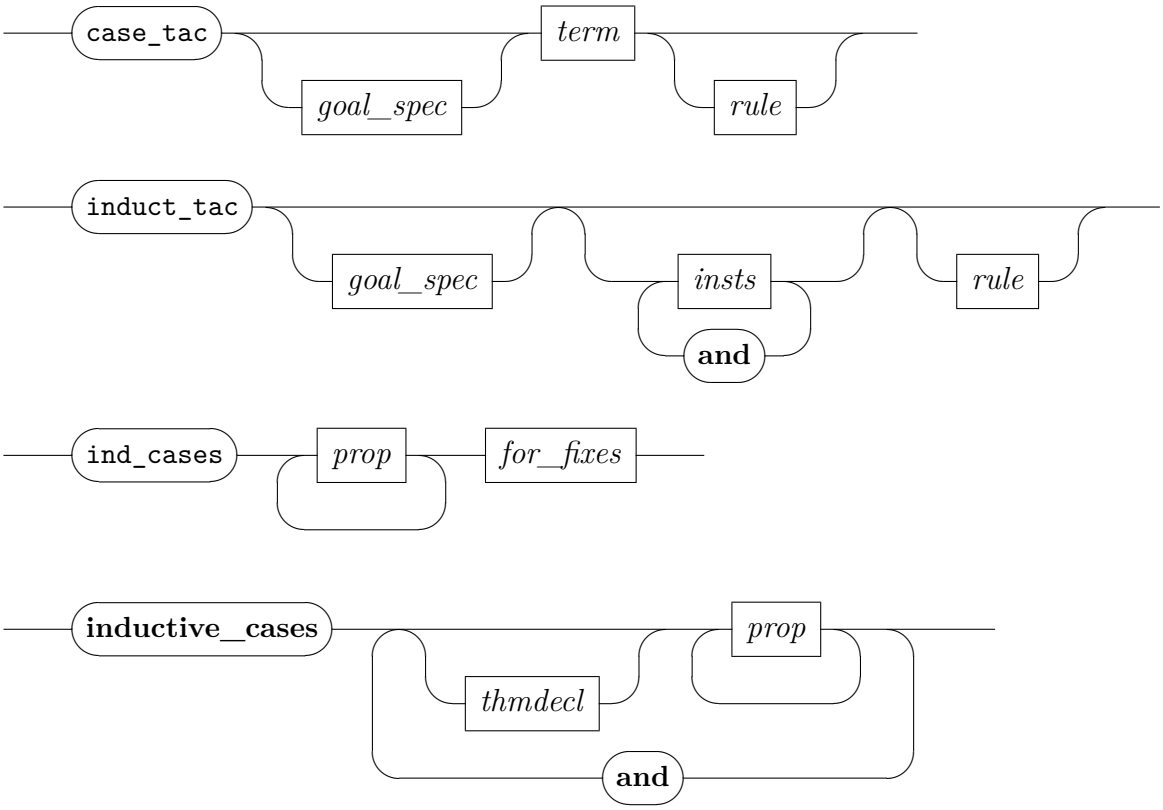


coherent solves problems of *Coherent Logic* [7], which covers applications in confluence theory, lattice theory and projective geometry. See `~~/src/HOL/Examples/Coherent.thy` for some examples.

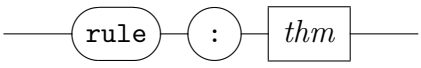
12.9 Unstructured case analysis and induction

The following tools of Isabelle/HOL support cases analysis and induction in unstructured tactic scripts; see also §6.5 for proper Isar versions of similar ideas.

case_tac* : *method*
induct_tac* : *method*
ind_cases* : *method*
inductive_cases* : *local_theory* → *local_theory*



rule



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. §6.5. The **datatype** package already takes care of this.

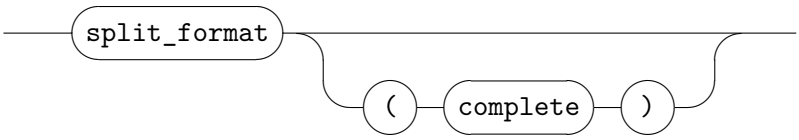
These unstructured tactics feature both goal addressing and dynamic instantiation. Note that named rule cases are *not* provided as would be by the proper *cases* and *induct* proof methods (see §6.5). Unlike the *induct* method, *induct_tac* does not handle structured rule statements, only the compact 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 *ind_cases* is a proof method to apply the result immediately as elimination rules, **inductive_cases** provides case split theorems at the theory level for later use. The **for** argument of the *ind_cases* method allows to specify a list of variables that should be generalized before applying the resulting rule.

12.10 Adhoc tuples

*split_format** : *attribute*



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.

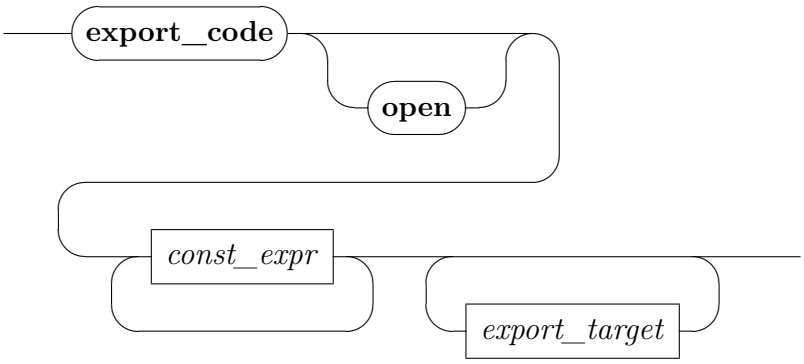
Chapter 13

Executable code

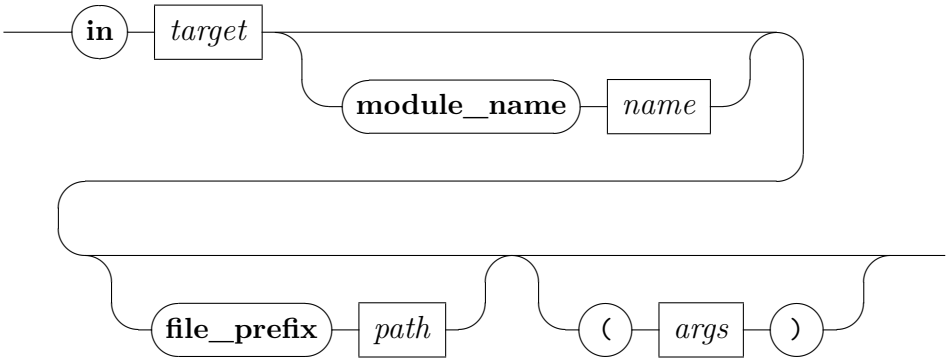
For validation purposes, it is often useful to *execute* specifications. In principle, execution could be simulated by Isabelle's inference kernel, i.e. by a combination of resolution and simplification. Unfortunately, this approach is rather inefficient. A more efficient way of executing specifications 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 [34], OCaml [29], Haskell [49] and Scala [15]. Conceptually, code generation is split up in three steps: *selection* of code theorems, *translation* into an abstract executable view and *serialization* to a specific *target language*. Inductive specifications can be executed using the predicate compiler which operates within HOL. See [21] for an introduction.

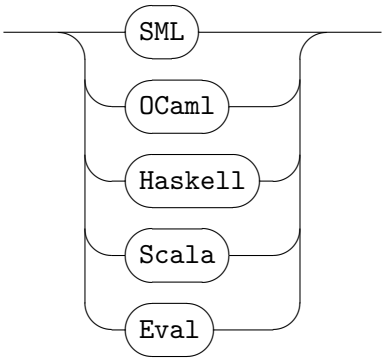
export_code*	:	*local_theory* → *local_theory*
code	:	*attribute*
code_datatype	:	*theory* → *theory*
print_codesetup*	:	*context* →
code_unfold	:	*attribute*
code_post	:	*attribute*
code_abbrev	:	*attribute*
print_codeproc*	:	*context* →
code_thms*	:	*context* →
code_deps*	:	*context* →
code_reserved	:	*theory* → *theory*
code_printing	:	*theory* → *theory*
code_identifier	:	*theory* → *theory*
code_monad	:	*theory* → *theory*
code_reflect	:	*theory* → *theory*
code_pred	:	*theory* → *proof*(*prove*)
code_timing	:	*attribute*
code_simp_trace	:	*attribute*
code_runtime_trace	:	*attribute*



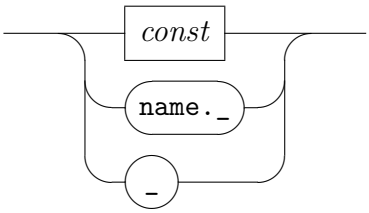
export_target



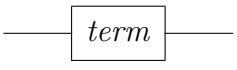
target



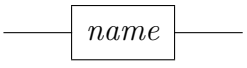
const_expr



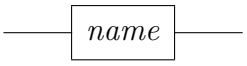
const



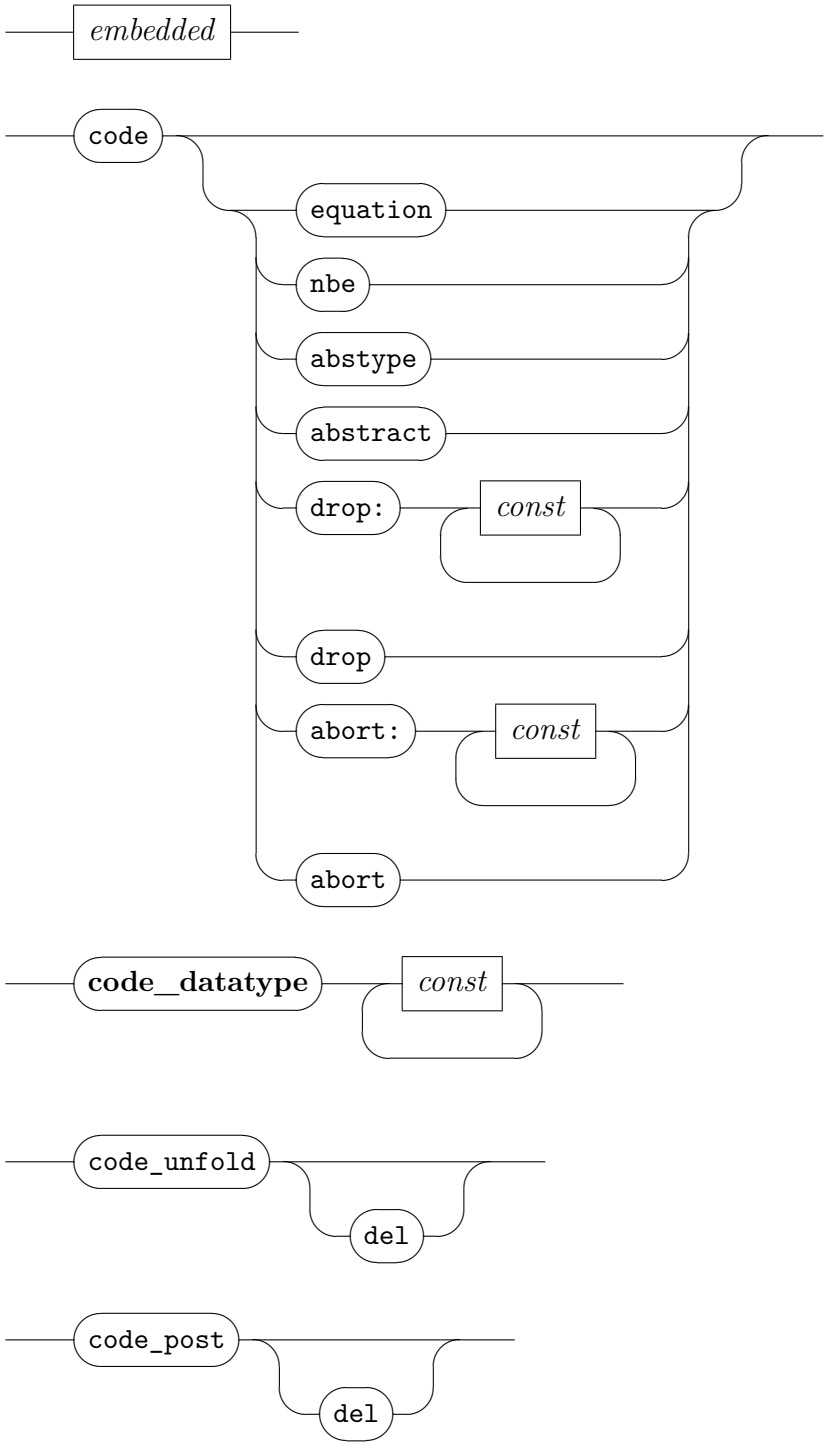
type_constructor

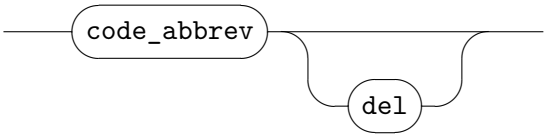


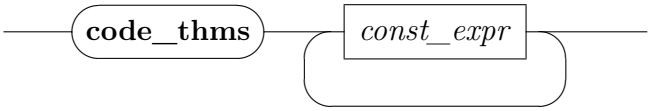
class

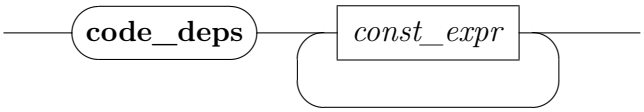


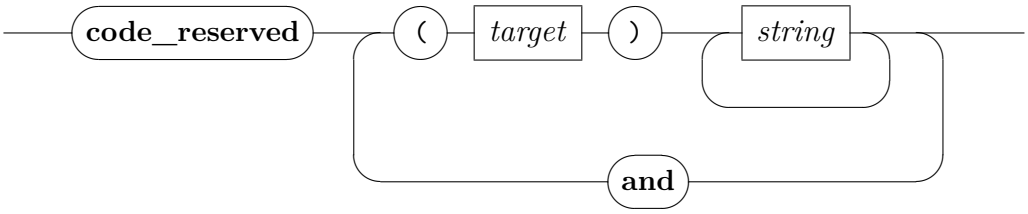
path


embedded
code
equation
nbe
abstype
abstract
drop:
const
drop
abort:
const
abort
code_datatype
const
code_unfold
del
code_post
del

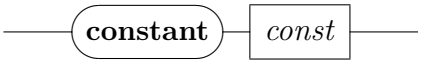




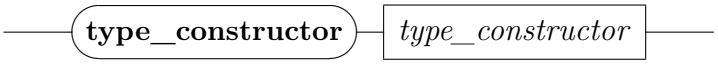




symbol_const



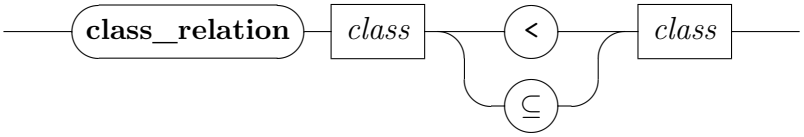
symbol_type_constructor



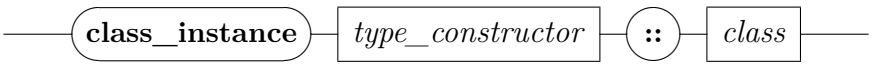
symbol_class



symbol_class_relation



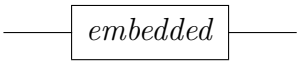
symbol_class_instance



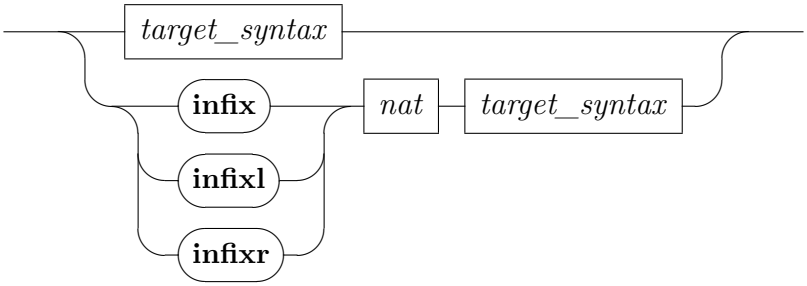
symbol_module



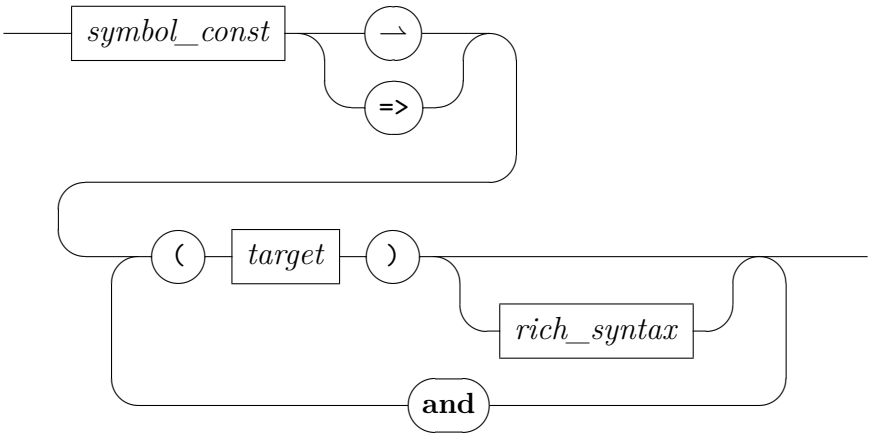
target_syntax



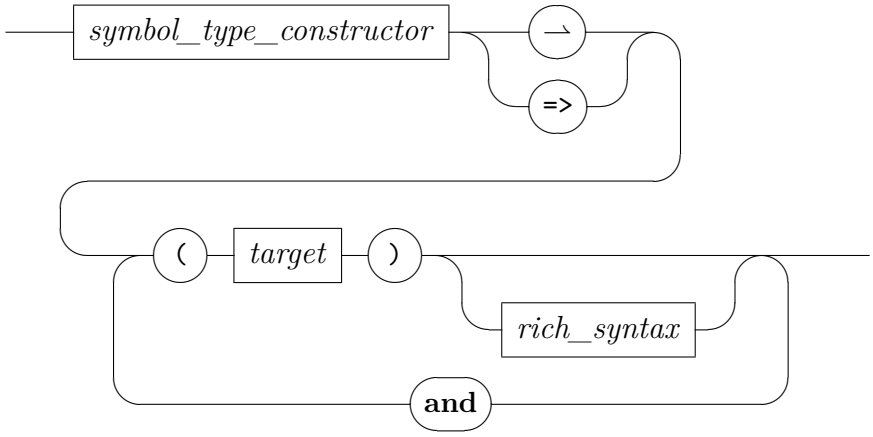
rich_syntax



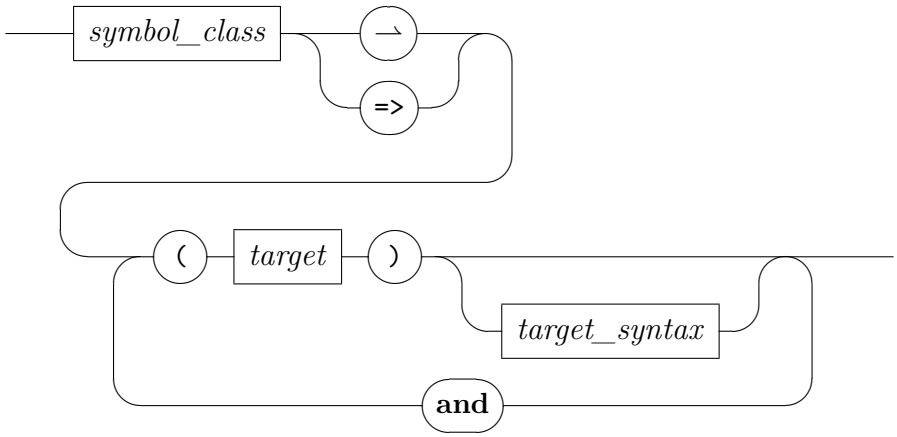
printing_const



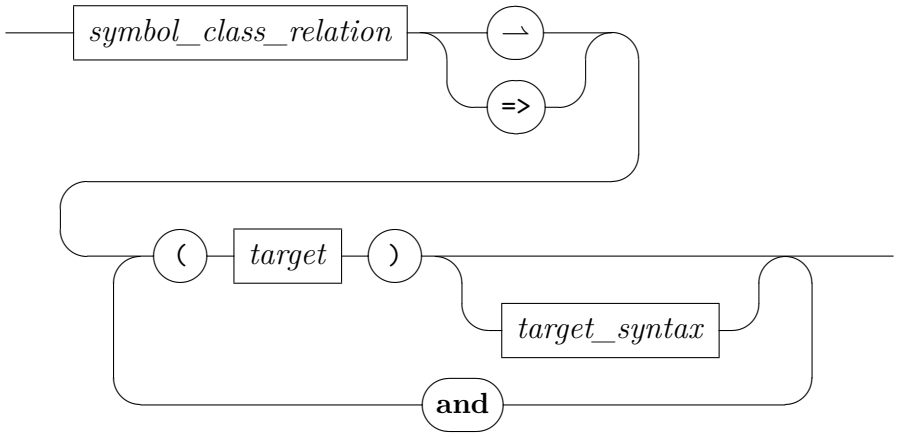
printing_type_constructor



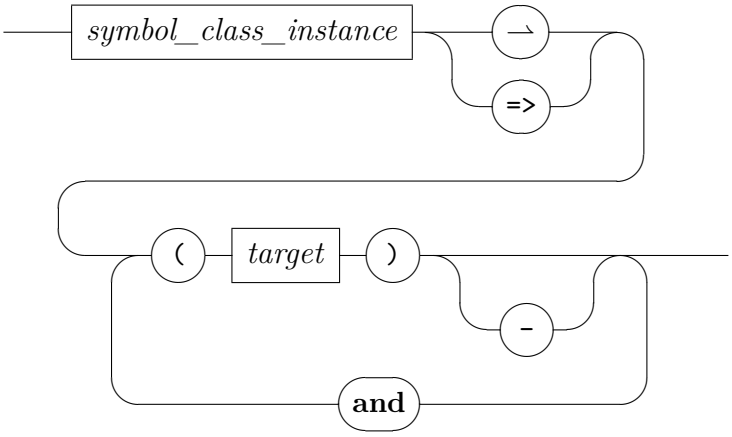
printing_class



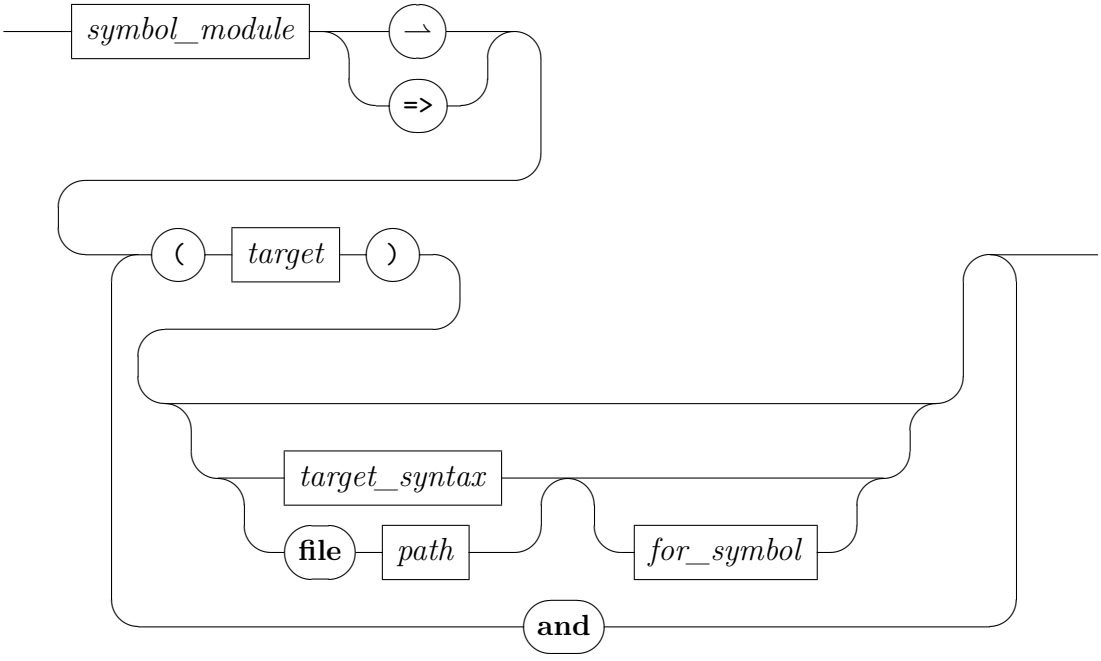
printing_class_relation



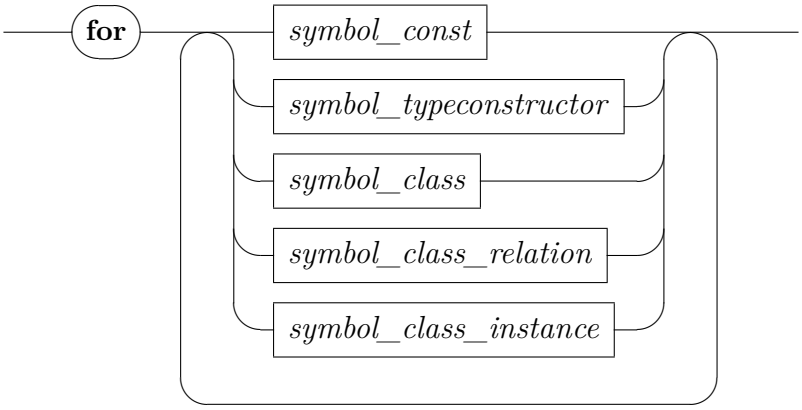
printing_class_instance

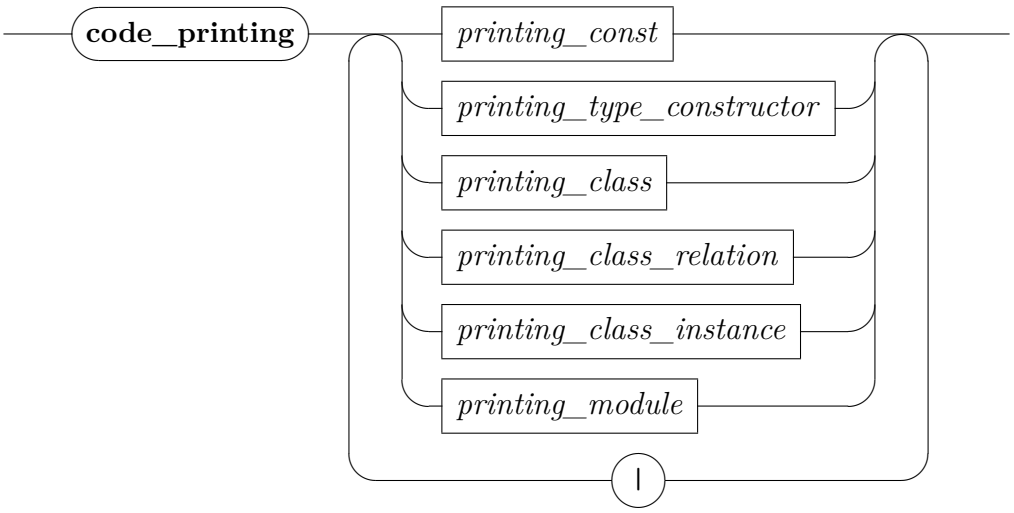


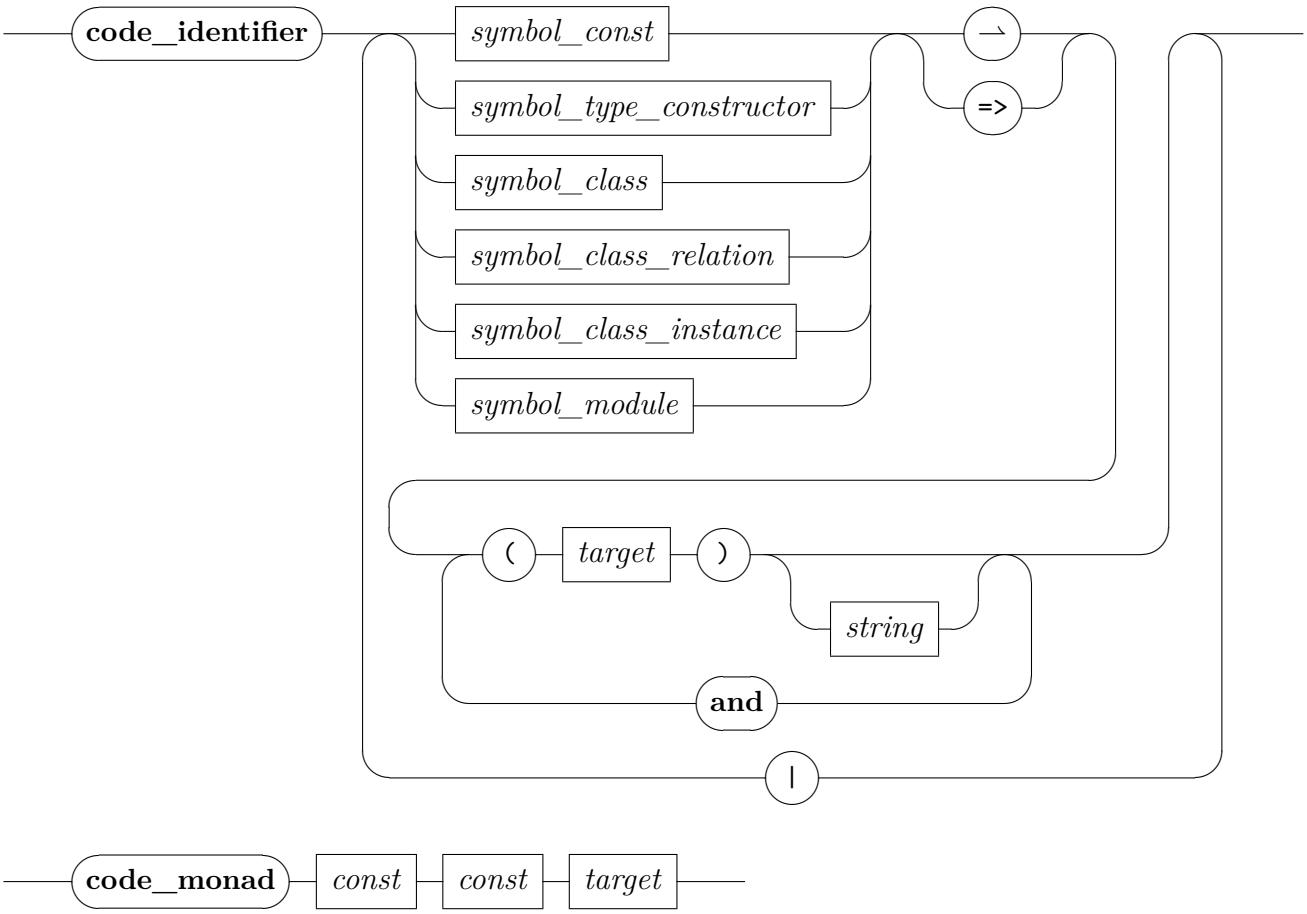
printing_module

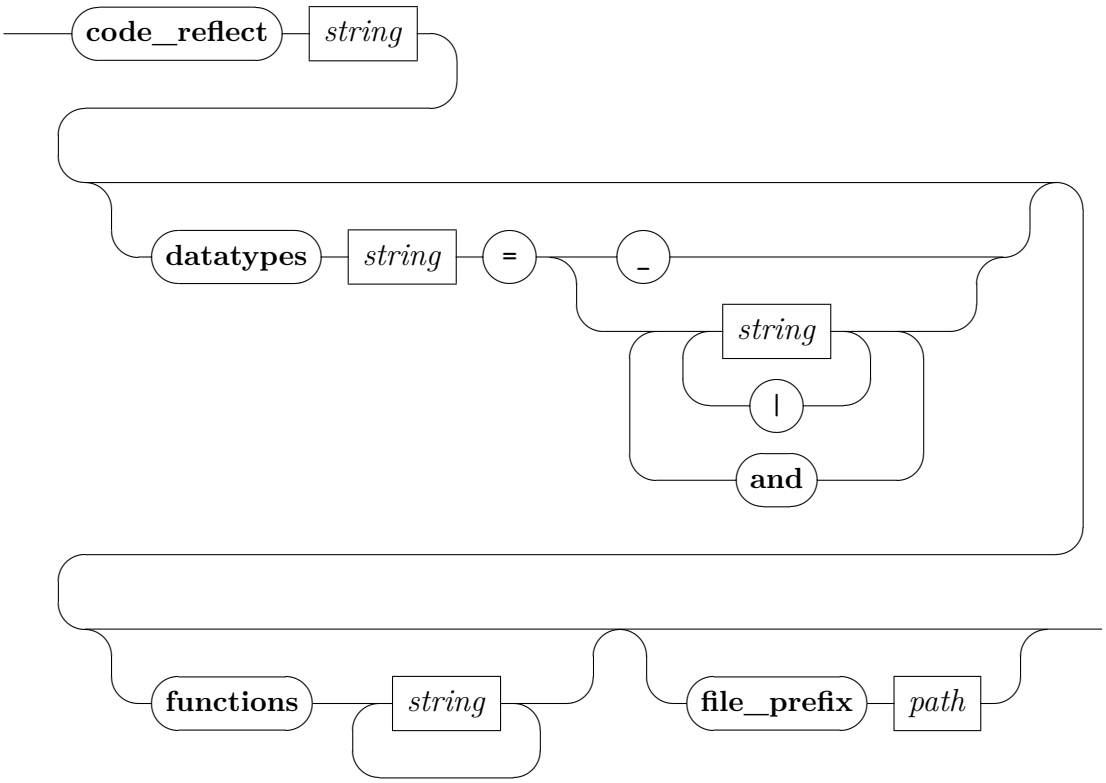


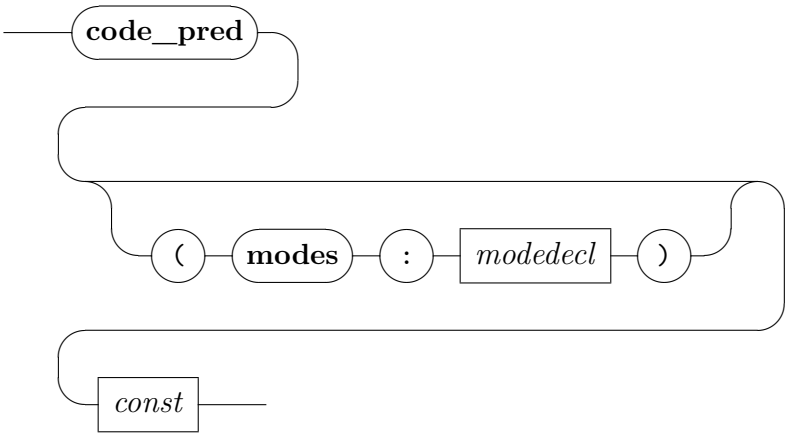
for_symbol


for
symbol_const
symbol_typeconstructor
symbol_class
symbol_class_relation
symbol_class_instance

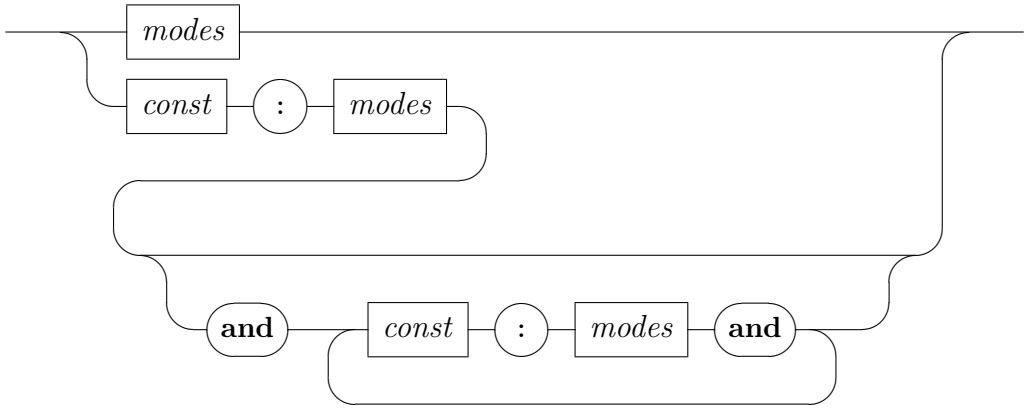

code_printing
printing_const
printing_type_constructor
printing_class
printing_class_relation
printing_class_instance
printing_module
|


code_identifier
symbol_const
symbol_type_constructor
symbol_class
symbol_class_relation
symbol_class_instance
symbol_module
→
=>
(
target
)
string
and
|
code_monad
const
const
target

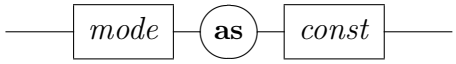

code_reflect
string
datatypes
string
=
_
string
|
and
functions
string
file_prefix
path


code_pred
(
modes
:
modedecl
)
const

modedecl



modes



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 constants within a certain theory by giving *name*._, or referring to *all* executable constants currently available by giving _.

By default, exported identifiers are minimized per module. This can be suppressed by prepending **open** before the list of constants.

By default, for each involved theory one corresponding name space module is generated. Alternatively, a module name may be specified after the **module_name** keyword; then *all* code is placed in this module.

Generated code is output as logical files within the theory context, as well as session exports that can be retrieved using `isabelle export`, or `isabelle build` with option `-e` and suitable **export_files** specifications in the session `ROOT` entry. All files have a common directory prefix: the long theory name plus "`code`". The actual file name is determined by the target language together with an optional **file_prefix** (the default is "`export`" with a consecutive number within the current theory). For *SML*, *OCaml* and *Scala*, the file prefix becomes a plain file with extension (e.g. "`.ML`" for SML). For *Haskell* the file prefix becomes a directory that is populated with a separate file for each module (with extension "`.hs`").

Serializers take an optional list of arguments in parentheses.

- 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.
- For *Scala*, "*case_insensitive*" avoids name clashes on case-insensitive file systems.

code declares code equations for code generation.

Variant *code equation* declares a conventional equation as code equation.

Variant *code nbe* accepts also non-left-linear equations for *normalization by evaluation* only.

Multiple *code equation* / *code nbe* declarations referring to the same constant within the same theory are handled as *one* function declaration for that particular constant: the first code declaration within a theory disregards any previous function declaration and superseedes any equations from preceeding theories.

Each code equation is prepended to existing code equations declared in the same theory, with syntactically subsumed equations removed.

Packages usually provide reasonable default code equations; an explicit declaration of a code equation superseedes any preceeding default code equations.

Variants *code abstype* and *code abstract* declare abstract datatype certificates or code equations on abstract datatype representations respectively.

Vanilla *code* falls back to *code equation* or *code abstract* depending on the syntactic shape of the underlying equation.

Variants *code drop*: and *code abort*: take a list of constants as arguments and drop all code equations declared for them. In the case of *abort*, these constants if needed are implemented by program abort (exception). Variants *code drop* and *code abort* derive the affected constants from the underlying theorems interpreted as code equations.

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 (or with option "*del*" removes) 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_reserved declares a list of names as reserved for a given target, preventing it to be shadowed by any generated code.

code_printing associates a series of symbols (constants, type constructors, classes, class relations, instances, module names) with target-specific serializations; omitting a serialization deletes an existing serialization.

code_monad provides an auxiliary mechanism to generate monadic code for Haskell.

code_identifier associates a a series of symbols (constants, type constructors, classes, class relations, instances, module names) with target-specific hints how these symbols shall be named. These hints gain precedence over names for symbols with no hints at all. Conflicting hints are subject to name disambiguation. *Warning:* It is at the discretion of the user to ensure that name prefixes of identifiers in compound statements like type classes or datatypes are still the same.

code_reflect without a "**file_prefix**" 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_prefix**" argument, the corresponding code is generated/exported to the specified file (as for **export_code**) 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.

code_timing scrapes timing samples from different stages of the code generator.

code_simp_trace traces the simplifier when it is used with code equations.

code_runtime_trace traces ML code generated dynamically for execution.

Part IV

Appendix

Appendix A

Isabelle/Isar quick reference

A.1 Proof commands

A.1.1 Main grammar

main = **notepad begin** *statement** **end**
| **theorem** *name*: *props* **if** *name*: *props* **for** *vars proof*
| **theorem** *name*:
fixes *vars*
assumes *name*: *props*
shows *name*: *props proof*
| **theorem** *name*:
fixes *vars*
assumes *name*: *props*
obtains (*name*) *vars* **where** *props* | ... *proof*

proof = *refinement** *proper_proof*

refinement = **apply** *method*
| **supply** *name* = *thms*
| **subgoal premises** *name* **for** *vars proof*
| **using** *thms*
| **unfolding** *thms*

proper_proof = **proof** *method*$^?$ *statement** **qed** *method*$^?$
| **done**

statement = **{** *statement** **}**
| **next**
| **note** *name* = *thms*
| **let** *term* = *term*
| **write** *name* (*mixfix*)
| **fix** *vars*
| **assume** *name*: *props* **if** *props* **for** *vars*
| **then**$^?$ *goal*

goal = **have** *name*: *props* **if** *name*: *props* **for** *vars proof*
| **show** *name*: *props* **if** *name*: *props* **for** *vars proof*

A.1.2 Primitives

fix x	augment context by $\bigwedge x.\ \square$
assume a: A	augment context by $A \Longrightarrow \square$
then	indicate forward chaining of facts
have a: A	prove local result
show a: A	prove local result, refining some goal
using a	indicate use of additional facts
unfolding a	unfold definitional equations
proof m_1 ... **qed** m_2	indicate proof structure and refinements
{ ... **}**	indicate explicit blocks
next	switch proof blocks
note $a = b$	reconsider and declare facts
let $p = t$	abbreviate terms by higher-order matching
write c (mx)	declare local mixfix syntax

A.1.3 Abbreviations and synonyms

by m_1 m_2	$\equiv$	**proof** m_1 **qed** m_2
..	$\equiv$	**by** *standard*
.	$\equiv$	**by** *this*
from a	$\equiv$	**note** a **then**
with a	$\equiv$	**from** a **and** *this*
from *this*	$\equiv$	**then**

A.1.4 Derived elements

also$_0$	$\approx$	**note** *calculation* = *this*
also$_{n+1}$	$\approx$	**note** *calculation* = *trans* [*OF calculation this*]
finally	$\approx$	**also from** *calculation*
moreover	$\approx$	**note** *calculation* = *calculation this*
ultimately	$\approx$	**moreover from** *calculation*
presume a: A	$\approx$	**assume** a: A
define x **where** $x = t$	$\approx$	**fix** x **assume** x_def: $x = t$
consider x **where** $A \mid \ldots$	$\approx$	**have** *thesis* **if** $\bigwedge x.\ A \Longrightarrow$ *thesis* **and** ... **for** *thesis*
obtain x **where** a: A $\langle proof \rangle$	$\approx$	**consider** x **where** A $\langle proof \rangle$ **fix** x **assume** a: A
case c	$\approx$	**fix** x **assume** c: A
sorry	$\approx$	**by** *cheating*

A.1.5 Diagnostic commands

typ τ	print type
term t	print term
prop φ	print proposition
thm a	print fact
print_statement a	print fact in long statement form

A.2 Proof methods

Single steps (forward-chaining facts)

assumption	apply some goal assumption
this	apply current facts
rule a	apply some rule
standard	apply standard rule (default for **proof**)
contradiction	apply ¬ 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
intro_locales	locale introduction rules (without body)
unfold_locales	locale introduction rules (with body)
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

	rule	*iprover*	*blast* *fast*	*simp* *simp_all*	*auto* *force*
Pure.elim! *Pure.intro*!	×	×			
Pure.elim Pure.intro	×	×			
elim! *intro*!	×		×		×
elim intro	×		×		×
iff	×		×	×	×
iff ?	×				
elim? intro?	×				
simp				×	×
cong				×	×
split				×	×

A.5 Proof scripts

A.5.1 Commands

apply *m*	apply proof method during backwards refinement
apply_end *m*	apply proof method (as if in terminal position)
supply *a*	supply facts during backwards refinement
subgoal	nested proof during backwards refinement
defer *n*	move subgoal to end
prefer *n*	move subgoal to start
back	backtrack last command
done	complete proof

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 φ	delete assumptions
subgoal_tac φ	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)

Appendix B

Predefined Isabelle symbols

Isabelle supports an infinite number of non-ASCII symbols, which are represented in source text as `\<`*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 `\isasym`*name* for each `\<`*name*`>` in the `isabellesym.sty` file. Most of these symbols are displayed properly in Isabelle/jEdit and LaTeX generated from Isabelle.

Moreover, any single symbol (or ASCII character) may be prefixed by `\<^sup>` for superscript and `\<^sub>` for subscript, such as `A\<^sup>\<star>` for $A^\star$ and `A\<^sub>1` for A_1. Sub- and superscripts that span a region of text can be marked up with `\<^bsub>`...`\<^esub>` and `\<^bsup>`...`\<^esup>` respectively, but note that there are limitations in the typographic rendering quality of this form. Furthermore, all ASCII characters and most other symbols may be printed in bold by prefixing `\<^bold>` such as `\<^bold>\<alpha>` for $\boldsymbol{\alpha}$. Note that `\<^sup>`, `\<^sub>`, `\<^bold>` cannot be combined.

Further details of Isabelle document preparation are covered in chapter 4.

`\<zero>`	$\mathbf{0}$	`\<one>`	$\mathbf{1}$
`\<two>`	$\mathbf{2}$	`\<three>`	$\mathbf{3}$
`\<four>`	$\mathbf{4}$	`\<five>`	$\mathbf{5}$
`\<six>`	$\mathbf{6}$	`\<seven>`	$\mathbf{7}$
`\<eight>`	$\mathbf{8}$	`\<nine>`	$\mathbf{9}$
`\<A>`	$\mathcal{A}$	`\<B>`	$\mathcal{B}$
`\<C>`	$\mathcal{C}$	`\<D>`	$\mathcal{D}$
`\<E>`	$\mathcal{E}$	`\<F>`	$\mathcal{F}$
`\<G>`	$\mathcal{G}$	`\<H>`	$\mathcal{H}$
`\<I>`	$\mathcal{I}$	`\<J>`	$\mathcal{J}$
`\<K>`	$\mathcal{K}$	`\<L>`	$\mathcal{L}$
`\<M>`	$\mathcal{M}$	`\<N>`	$\mathcal{N}$
`\<O>`	$\mathcal{O}$	`\<P>`	$\mathcal{P}$
`\<Q>`	$\mathcal{Q}$	`\<R>`	$\mathcal{R}$

\<S>	$\mathcal{S}$	\<T>	$\mathcal{T}$
\<U>	$\mathcal{U}$	\<V>	$\mathcal{V}$
\<W>	$\mathcal{W}$	\<X>	$\mathcal{X}$
\<Y>	$\mathcal{Y}$	\<Z>	$\mathcal{Z}$
\<a>	a	\<b>	b
\<c>	c	\<d>	d
\<e>	e	\<f>	f
\<g>	g	\<h>	h
\<i>	i	\<j>	j
\<k>	k	\<l>	l
\<m>	m	\<n>	n
\<o>	o	\<p>	p
\<q>	q	\<r>	r
\<s>	s	\<t>	t
\<u>	u	\<v>	v
\<w>	w	\<x>	x
\<y>	y	\<z>	z
\<AA>	$\mathfrak{A}$	\<BB>	$\mathfrak{B}$
\<CC>	$\mathfrak{C}$	\<DD>	$\mathfrak{D}$
\<EE>	$\mathfrak{E}$	\<FF>	$\mathfrak{F}$
\<GG>	$\mathfrak{G}$	\<HH>	$\mathfrak{H}$
\<II>	$\mathfrak{I}$	\<JJ>	$\mathfrak{J}$
\<KK>	$\mathfrak{K}$	\<LL>	$\mathfrak{L}$
\<MM>	$\mathfrak{M}$	\<NN>	$\mathfrak{N}$
\<OO>	$\mathfrak{O}$	\<PP>	$\mathfrak{P}$
\<QQ>	$\mathfrak{Q}$	\<RR>	$\mathfrak{R}$
\<SS>	$\mathfrak{S}$	\<TT>	$\mathfrak{T}$
\<UU>	$\mathfrak{U}$	\<VV>	$\mathfrak{V}$
\<WW>	$\mathfrak{W}$	\<XX>	$\mathfrak{X}$
\<YY>	$\mathfrak{Y}$	\<ZZ>	$\mathfrak{Z}$
\<aa>	$\mathfrak{a}$	\<bb>	$\mathfrak{b}$
\<cc>	$\mathfrak{c}$	\<dd>	$\mathfrak{d}$
\<ee>	$\mathfrak{e}$	\<ff>	$\mathfrak{f}$
\<gg>	$\mathfrak{g}$	\<hh>	$\mathfrak{h}$
\<ii>	$\mathfrak{i}$	\<jj>	$\mathfrak{j}$
\<kk>	$\mathfrak{k}$	\<ll>	$\mathfrak{l}$
\<mm>	$\mathfrak{m}$	\<nn>	$\mathfrak{n}$
\<oo>	$\mathfrak{o}$	\<pp>	$\mathfrak{p}$
\<qq>	$\mathfrak{q}$	\<rr>	$\mathfrak{r}$
\<ss>	$\mathfrak{s}$	\<tt>	$\mathfrak{t}$
\<uu>	$\mathfrak{u}$	\<vv>	$\mathfrak{v}$
\<ww>	$\mathfrak{w}$	\<xx>	$\mathfrak{x}$

\<yy>	$\mathfrak{y}$	\<zz>	$\mathfrak{z}$
\<alpha>	α	\<beta>	β
\<gamma>	γ	\<delta>	δ
\<epsilon>	ε	\<zeta>	ζ
\<eta>	η	\<theta>	ϑ
\<iota>	ι	\<kappa>	κ
\<lambda>	λ	\<mu>	μ
\<nu>	ν	\<xi>	ξ
\<pi>	π	\<rho>	ϱ
\<sigma>	σ	\<tau>	τ
\<upsilon>	υ	\<phi>	φ
\<chi>	χ	\<psi>	ψ
\<omega>	ω	\<Gamma>	Γ
\<Delta>	Δ	\<Theta>	Θ
\<Lambda>	Λ	\<Xi>	Ξ
\<Pi>	Π	\<Sigma>	Σ
\<Upsilon>	Υ	\<Phi>	Φ
\<Psi>	Ψ	\<Omega>	Ω
\<bbbA>	$\mathbb{A}$	\<bool>	$\mathbb{B}$
\<complex>	$\mathbb{C}$	\<bbbD>	$\mathbb{D}$
\<bbbE>	$\mathbb{E}$	\<bbbF>	$\mathbb{F}$
\<bbbG>	$\mathbb{G}$	\<bbbH>	$\mathbb{H}$
\<bbbI>	$\mathbb{I}$	\<bbbJ>	$\mathbb{J}$
\<bbbK>	$\mathbb{K}$	\<bbbL>	$\mathbb{L}$
\<bbbM>	$\mathbb{M}$	\<nat>	$\mathbb{N}$
\<bbbO>	$\mathbb{O}$	\<bbbP>	$\mathbb{P}$
\<rat>	$\mathbb{Q}$	\<real>	$\mathbb{R}$
\<bbbS>	$\mathbb{S}$	\<bbbT>	$\mathbb{T}$
\<bbbU>	$\mathbb{U}$	\<bbbV>	$\mathbb{V}$
\<bbbW>	$\mathbb{W}$	\<bbbX>	$\mathbb{X}$
\<bbbY>	$\mathbb{Y}$	\<int>	$\mathbb{Z}$
\<leftarrow>	$\leftarrow$	\<rightarrow>	$\rightarrow$
\<longleftarrow>	$\longleftarrow$	\<longrightarrow>	$\longrightarrow$
\<longlongleftarrow>	$\longleftarrow$	\<longlongrightarrow>	$\longrightarrow$
\<longlonglongleftarrow>	$\longleftarrow$	\<longlonglongrightarrow>	$\longrightarrow$
\<Leftarrow>	$\Leftarrow$	\<Rightarrow>	$\Rightarrow$
\<Longleftarrow>	$\Longleftarrow$	\<Longrightarrow>	$\Longrightarrow$
\<Lleftarrow>	$\Lleftarrow$	\<Rrightarrow>	$\Rrightarrow$
\<leftrightarrow>	$\leftrightarrow$	\<Leftrightarrow>	$\Leftrightarrow$
\<longleftrightarrow>	$\longleftrightarrow$	\<Longleftrightarrow>	$\Longleftrightarrow$
\<mapsto>	$\mapsto$	\<longmapsto>	$\longmapsto$
\<midarrow>	$-$	\<Midarrow>	$=$

\<hookleftarrow>	↩	\<hookrightarrow>	↪
\<leftharpoondown>	↽	\<rightharpoondown>	⇁
\<leftharpoonup>	↼	\<rightharpoonup>	⇀
\<rightleftharpoons>	⇌	\<leadsto>	⇝
\<downharpoonleft>	⇃	\<downharpoonright>	⇂
\<upharpoonleft>	↿	\<upharpoonright>	↾
\<restriction>	↾	\<Colon>	::
\<up>	↑	\<Up>	⇑
\<down>	↓	\<Down>	⇓
\<updown>	↕	\<Updown>	⇕
\<langle>	⟨	\<rangle>	⟩
\<llangle>	⦇	\<rrangle>	⦈
\<lceil>	⌈	\<rceil>	⌉
\<lfloor>	⌊	\<rfloor>	⌋
\<lparr>	⦅	\<rparr>	⦆
\<lbrakk>	⟦	\<rbrakk>	⟧
\<lbrace>	⦃	\<rbrace>	⦄
\<lblot>	⦉	\<rblot>	⦊
\<guillemotleft>	«	\<guillemotright>	»
\<bottom>	⊥	\<top>	⊤
\<and>	∧	\<And>	⋀
\<or>	∨	\<Or>	⋁
\<forall>	∀	\<exists>	∃
\<not>	¬	\<nexists>	∄
\<circle>	○	\<box>	□
\<diamond>	◇	\<diamondop>	⋄
\<surd>	√	\<turnstile>	⊢
\<Turnstile>	⊨	\<tturnstile>	⊩
\<TTurnstile>	⊫	\<stileturn>	⊣
\<le>	≤	\<ge>	≥
\<lless>	≪	\<ggreater>	≫
\<lesssim>	≲	\<greatersim>	≳
\<lessapprox>	≲	\<greaterapprox>	≳
\<in>	∈	\<notin>	∉
\<subset>	⊂	\<supset>	⊃
\<subseteq>	⊆	\<supseteq>	⊇
\<sqsubset>	⊏	\<sqsupset>	⊐
\<sqsubseteq>	⊑	\<sqsupseteq>	⊒
\<inter>	∩	\<Inter>	⋂
\<union>	∪	\<Union>	⋃
\<squnion>	⊔	\<Squnion>	⨆
\<sqinter>	⊓	\<Sqinter>	⨅

\<setminus>	$\setminus$	\<propto>	$\propto$
\<uplus>	$\uplus$	\<Uplus>	$\biguplus$
\<noteq>	$\neq$	\<sim>	$\sim$
\<doteq>	$\doteq$	\<simeq>	$\simeq$
\<approx>	$\approx$	\<asymp>	$\asymp$
\<cong>	$\cong$	\<smile>	$\smile$
\<equiv>	$\equiv$	\<frown>	$\frown$
\<Join>	$\bowtie$	\<bowtie>	$\bowtie$
\<prec>	$\prec$	\<succ>	$\succ$
\<preceq>	$\preceq$	\<succeq>	$\succeq$
\<parallel>	$\parallel$	\<Parallel>	$\parallel$
\<interleace>	$\|\|\|$	\<sslash>	$/\!/$
\<bar>	$\mid$	\<bbar>	$[\!]$
\<plusminus>	$\pm$	\<minusplus>	$\mp$
\<times>	$\times$	\	$\div$
\<cdot>	$\cdot$	\<sqdot>	$\centerdot$
\<star>	$\star$	\<bullet>	$\bullet$
\<circ>	$\circ$	\<dagger>	$\dagger$
\<ddagger>	$\ddagger$	\<lhd>	$\lhd$
\<rhd>	$\rhd$	\<unlhd>	$\unlhd$
\<unrhd>	$\unrhd$	\<triangleleft>	$\triangleleft$
\<triangleright>	$\triangleright$	\<triangle>	$\triangle$
\<triangleq>	$\triangleq$	\<oplus>	$\oplus$
\<Oplus>	$\bigoplus$	\<otimes>	$\otimes$
\<Otimes>	$\bigotimes$	\<odot>	$\odot$
\<Odot>	$\bigodot$	\<ominus>	$\ominus$
\<oslash>	$\oslash$	\<dots>	$\dots$
\<cdots>	$\cdots$	\<Sum>	$\sum$
\<Prod>	$\prod$	\<Coprod>	$\coprod$
\<infinity>	∞	\<integral>	$\int$
\<ointegral>	$\oint$	\<clubsuit>	$\clubsuit$
\<diamondsuit>	$\diamondsuit$	\<heartsuit>	$\heartsuit$
\<spadesuit>	$\spadesuit$	\<aleph>	$\aleph$
\<emptyset>	$\emptyset$	\<nabla>	∇
\<partial>	∂	\<Re>	$\Re$
\<Im>	$\Im$	\<flat>	$\flat$
\<natural>	$\natural$	\<sharp>	$\sharp$
\<angle>	$\angle$	\<copyright>	©
\<registered>	®	\<inverse>	$^{-1}$
\<onequarter>	¼	\<onehalf>	½
\<threequarters>	¾	\<ordfeminine>	^{a}
\<ordmasculine>	^{o}	\<section>	§

\<paragraph>	¶	\<exclamdown>	¡
\<questiondown>	¿	\<euro>	€
\<pounds>	£	\<yen>	¥
\<cent>	¢	\<currency>	¤
\<degree>	°	\<hyphen>	-
\<amalg>	∐	\<mho>	℧
\<lozenge>	◊	\<wp>	℘
\<wrong>	≀	\<acute>	´
\<index>	ı	\<dieresis>	¨
\<cedilla>	¸	\<hungarumlaut>	˝
\<some>	ϵ	\<bind>	≫=
\<then>	≫	\<Zcomp>	⨟
\<Zinj>	⟼	\<Zpinj>	⤔
\<Zfinj>	⤕	\<Zsurj>	↠
\<Zpsurj>	⤀	\<Zbij>	⤖
\<Zpfun>	⇸	\<Zffun>	⇻
\<Zdres>	◁	\<Zndres>	⩤
\<Zrres>	▷	\<Znrres>	⩥
\<Zspot>	•	\<Zproject>	⨡
\<Zsemi>	⨟	\<Ztypecolon>	∶
\<Zhide>	⧹	\<Zcat>	⌢
\<Zinbag>	⋵	\<hole>	¤
\<newline>	⏎	\<comment>	—
\<proof>	⟨*proof*⟩	\<open>	‹
\<close>	›	\<checkmark>	✓
\<crossmark>	✗		

Bibliography

[1] P. Andrews. *An Introduction to Mathematical Logic and Type Theory: to Truth through Proof*. Computer Science and Applied Mathematics. Academic Press, 1986.

[2] D. Aspinall. Proof General. http://proofgeneral.inf.ed.ac.uk/.

[3] D. Aspinall. Proof General: A generic tool for proof development. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1785 of *Lecture Notes in Computer Science*, pages 38–42. Springer-Verlag, 2000.

[4] C. Ballarin. Locales: A module system for mathematical theories. *Journal of Automated Reasoning*, 52(2):123–153, 2014.

[5] G. Bauer and M. Wenzel. Calculational reasoning revisited — an Isabelle/Isar experience. In R. J. Boulton and P. B. Jackson, editors, *Theorem Proving in Higher Order Logics: TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[6] S. Berghofer and T. Nipkow. Proof terms for simply typed higher order logic. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 38–52. Springer-Verlag, 2000.

[7] M. Bezem and T. Coquand. Automating Coherent Logic. In G. Sutcliffe and A. Voronkov, editors, *LPAR-12*, volume 3835 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

[8] J. Biendarra, J. C. Blanchette, M. Desharnais, L. Panny, A. Popescu, and D. Traytel. *Defining (Co)datatypes and Primitively (Co)recursive Functions in Isabelle/HOL*. https://isabelle.in.tum.de/doc/datatypes.pdf.

[9] J. C. Blanchette. *Hammering Away: A User's Guide to Sledgehammer for Isabelle/HOL*. https://isabelle.in.tum.de/doc/sledgehammer.pdf.

[10] J. C. Blanchette. *Picking Nits: A User's Guide to Nitpick for Isabelle/HOL*. https://isabelle.in.tum.de/doc/nitpick.pdf.

[11] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*. Academic Press, 1988.

[12] A. Chaieb. *Automated methods for formal proofs in simple arithmetics and algebra*. PhD thesis, Technische Universität München, 2008. http://www4.in.tum.de/~chaieb/pubs/pdf/diss.pdf.

[13] A. Chaieb and M. Wenzel. Context aware calculation and deduction — ring equalities via Gröbner Bases in Isabelle. In M. Kauers, M. Kerber, R. Miner, and W. Windsteiger, editors, *Towards Mechanized Mathematical Assistants (CALCULEMUS 2007)*, volume 4573 of *LNAI*. Springer-Verlag, 2007.

[14] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[15] M. O. et al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[16] W. M. Farmer. The seven virtues of simple type theory. *J. Applied Logic*, 6(3):267–286, 2008.

[17] K. Futatsugi, J. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Symposium on Principles of Programming Languages*, pages 52–66, 1985.

[18] G. Gentzen. Untersuchungen über das logische Schließen. *Math. Zeitschrift*, 1935.

[19] M. J. C. Gordon. HOL: A machine oriented formulation of higher order logic. Technical Report 68, University of Cambridge Computer Laboratory, 1985.

[20] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[21] F. Haftmann. *Code generation from Isabelle theories*. https://isabelle.in.tum.de/doc/codegen.pdf.

[22] F. Haftmann. *Haskell-style type classes with Isabelle/Isar*. https://isabelle.in.tum.de/doc/classes.pdf.

[23] F. Haftmann and M. Wenzel. Constructive type classes in Isabelle. In T. Altenkirch and C. McBride, editors, *Types for Proofs and Programs, TYPES 2006*, volume 4502 of *LNCS*. Springer, 2007.

[24] B. Huffman and O. Kunčar. Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL. In *Certified Programs and Proofs (CPP 2013)*, volume 8307 of *Lecture Notes in Computer Science*. Springer-Verlag, 2013.

[25] A. Krauss. *Defining Recursive Functions in Isabelle/HOL.* https://isabelle.in.tum.de/doc/functions.pdf.

[26] A. Krauss. *Automating Recursive Definitions and Termination Proofs in Higher-Order Logic.* PhD thesis, Institut für Informatik, Technische Universität München, Germany, 2009.

[27] O. Kuncar. Correctness of Isabelle's cyclicity checker: Implementability of overloading in proof assistants. In *Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015, Mumbai, India, January 15-17, 2015*, 2015.

[28] O. Kuncar and A. Popescu. A consistent foundation for Isabelle/HOL. In C. Urban and X. Zhang, editors, *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, volume 9236 of *Lecture Notes in Computer Science*. Springer, 2015.

[29] X. Leroy et al. *The Objective Caml system – Documentation and user's manual.* http://caml.inria.fr/pub/docs/manual-ocaml/.

[30] D. W. Loveland. *Automated Theorem Proving: A Logical Basis.* North-Holland Publishing Co., 1978.

[31] U. Martin and T. Nipkow. Ordered rewriting and confluence. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, LNAI 449, pages 366–380. Springer, 1990.

[32] D. Matichuk, M. Wenzel, and T. C. Murray. An Isabelle proof method language. In G. Klein and R. Gamboa, editors, *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria*, volume 8558 of *LNCS*. Springer, 2014.

[33] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4), 1991.

[34] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[35] W. Naraschewski and M. Wenzel. Object-oriented verification based on record subtyping in higher-order logic. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics: TPHOLs '98*, volume 1479 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[36] T. Nipkow. Functional unification of higher-order patterns. In M. Vardi, editor, *Eighth Annual Symposium on Logic in Computer Science*, pages 64–74. IEEE Computer Society Press, 1993.

[37] T. Nipkow. Structured Proofs in Isar/HOL. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs (TYPES 2002)*, volume 2646 of *Lecture Notes in Computer Science*, pages 259–278. Springer-Verlag, 2003.

[38] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer, 2002. LNCS 2283.

[39] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[40] T. Nipkow and C. Prehofer. Type reconstruction for type classes. *Journal of Functional Programming*, 5(2):201–224, 1995.

[41] D. C. Oppen. Pretty printing. *ACM Transactions on Programming Languages and Systems*, 2(4), 1980.

[42] L. C. Paulson. *Isabelle's Logics*. https://isabelle.in.tum.de/doc/logics.pdf.

[43] L. C. Paulson. *Isabelle's Logics: FOL and ZF*. https://isabelle.in.tum.de/doc/logics-ZF.pdf.

[44] L. C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[45] L. C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

[46] L. C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[47] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996. https://www.cl.cam.ac.uk/~lp15/MLbook.

[48] F. J. Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, 2:191–216, 1986. Errata, JAR 4 (1988), 235–236 and JAR 18 (1997), 135.

[49] S. Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. http://www.haskell.org/definition/.

[50] A. Pitts. The HOL logic. In M. J. C. Gordon and T. F. Melham, editors, *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*, pages 191–232. Cambridge University Press, 1993.

[51] C. Runciman, M. Naylor, and F. Lindblad. Smallcheck and Lazy Smallcheck: Automatic exhaustive testing for small values. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell (Haskell 2008)*, pages 37–48. ACM, 2008.

[52] P. Schroeder-Heister. A natural extension of natural deduction. *Journal of Symbolic Logic*, 49(4), 1984.

[53] D. Traytel, S. Berghofer, and T. Nipkow. Extending Hindley-Milner Type Inference with Coercive Structural Subtyping. In H. Yang, editor, *APLAS 2011*, volume 7078 of *Lecture Notes in Computer Science*, pages 89–104, 2011.

[54] M. Wenzel. *The Isabelle System Manual*. https://isabelle.in.tum.de/doc/system.pdf.

[55] M. Wenzel. *The Isabelle/Isar Implementation*. https://isabelle.in.tum.de/doc/implementation.pdf.

[56] M. Wenzel. *Isabelle/jEdit*. https://isabelle.in.tum.de/doc/jedit.pdf.

[57] M. Wenzel. Type classes and overloading in higher-order logic. In E. L. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics: TPHOLs '97*, volume 1275 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[58] M. Wenzel. Isar — a generic interpretative approach to readable formal proof documents. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics: TPHOLs '99*, volume 1690 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[59] M. Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, Institut für Informatik, Technische Universität München, 2002. https://mediatum.ub.tum.de/doc/601724/601724.pdf.

[60] M. Wenzel. Isabelle/Isar — a generic framework for human-readable proof documents. In R. Matuszewski and A. Zalewska, editors, *From Insight to Proof — Festschrift in Honour of Andrzej Trybulec*, volume 10(23) of *Studies in Logic, Grammar, and Rhetoric*. University of Białystok, 2007. http://www.in.tum.de/~wenzelm/papers/isar-framework.pdf.

[61] M. Wenzel. Isabelle/jEdit — a Prover IDE within the PIDE framework. In J. Jeuring et al., editors, *Conference on Intelligent Computer Mathematics (CICM 2012)*, volume 7362 of *LNAI*. Springer, 2012.

[62] M. Wenzel and L. C. Paulson. Isabelle/Isar. In F. Wiedijk, editor, *The Seventeen Provers of the World*, LNAI 3600. Springer-Verlag, 2006.

[63] F. Wiedijk. Mizar: An impression. Unpublished paper, 1999. http://www.cs.kun.nl/~freek/mizar/mizarintro.ps.gz.

Index

- (method), **153**
. (command), **150**
.. (command), **150**
?case (variable), 155
?thesis (variable), 146
_ (fact), 142
{ (command), **136**
} (command), **136**

abbrev (antiquotation), **73**
abbreviation (command), **100**, 261
abbrevs (keyword), **95**
abs_def (attribute), 215, **216**
abstraction (notation kind), **188**
`addafter` (ML infix), **252**
`addbefore` (ML infix), **252**
`addSafter` (ML infix), **252**
`addSbefore` (ML infix), **252**
`addss` (ML), **252**
`addSss` (ML), **252**
`addSWrapper` (ML infix), **252**
`addWrapper` (ML infix), **252**
adhoc_overloading (command), **121**
algebra (HOL attribute), **307**
algebra (HOL method), **307**
alias (command), **133**
also (command), **146**
altstring (syntax), **54**, 54, 64
and (keyword), 63, 138
antiquotation (syntax), **75**
antiquotation_body (syntax), **76**
any (inner syntax), **193**, 195
application (notation kind), **188**
apply (command), 141, 142, **170**
apply_end (command), **170**
aprop (inner syntax), **194**, 195
args (syntax), **64**
arith (HOL attribute), **306**
arith (HOL method), **306**
arity (syntax), **59**
assms (fact), 143
assume (command), **137**
assumes (element), 106
assumption (inference), **32**
assumption (method), **153**
atom (syntax), **63**
atomize (attribute), **253**
atomize (method), **253**
attribute_setup (command), **122**
attributes (syntax), **64**
auto (method), **246**
axiomatization (command), **102**, 131
axmdecl (syntax), **65**

back (command), **170**
bash_function (antiquotation), **74**
best (method), **246**
bestsimp (method), **246**
binder (keyword), **189**
binder (notation kind), **188**
blast (method), **246**
bold (antiquotation), **74**
break (antiquotation option), **82**
build (tool), 71
bundle (antiquotation), **73**
bundle (command), **97**
by (command), **150**

calculation (fact), 146
cartouche (antiquotation option), **82**

cartouche (antiquotation), **74**
cartouche (inner syntax), **192**
cartouche (syntax), **54**, 55, 64, 75
case (command), 155, **157**
case_conclusion (attribute), **157**
case_names (attribute), **157**, 168
case_tac (HOL method), **310**
cases (attribute), **165**
cases (method), 159, **160**
chapter (command), **71**
cite (antiquotation), **74**
citep (antiquotation), **74**
citet (antiquotation), **74**
clamod (syntax), **247**
clarify (method), **250**
clarify_step (method), **251**
clarsimp (method), **250**
clasimpmod (syntax), **248**
class (antiquotation), **73**
class (command), **115**
class_deps (command), **115**
class_name (inner syntax), **195**
class_syntax (ML antiquotation), **207**
classdecl (syntax), **58**
cleaning (HOL method), **294**
code (HOL attribute), **313**
code_abbrev (HOL attribute), **313**
code_datatype (HOL command), **313**
code_deps (HOL command), **313**
code_identifier (HOL command), **313**
code_monad (HOL command), **313**
code_post (HOL attribute), **313**
code_pred (HOL command), **313**
code_printing (HOL command), **313**
code_reflect (HOL command), **313**
code_reserved (HOL command), **313**
code_runtime_trace (HOL attribute), **313**
code_simp_trace (HOL attribute), **313**
code_thms (HOL command), **313**
code_timing (HOL attribute), **313**
code_unfold (HOL attribute), **313**
coercion (HOL attribute), **304**
coercion_args (HOL attribute), **304**
coercion_delete (HOL attribute), **304**
coercion_enabled (HOL attribute), **304**
coercion_map (HOL attribute), **304**
coherent (HOL method), **309**
coinduct (attribute), **165**
coinduct (method), **160**
coinductive (HOL command), **260**
coinductive_set (HOL command), **260**
compile_generated_files (command), **126**
cong (attribute), **224**
consider (command), **166**
const (antiquotation), **73**
const_syntax (ML antiquotation), **207**
constrains (element), 106
consts (command), **119**
consumes (attribute), **157**
context (command), **95**, 160
context_elem (syntax), **108**
contradiction (method), **245**
contributor (document marker), **85**
corollary (command), **142**
creator (document marker), **85**
cut_tac (method), **175**

datatype (HOL command), 265
date (document marker), **85**
declaration (command), **103**
declare (command), **103**
deepen (method), **246**

default_sort (command), **130**
defer (command), **170**
define (command), **137**
defines (element), 106
definition (command), **100**
defn (attribute), **100**
`delSWrapper` (ML infix), **252**
`delWrapper` (ML infix), **252**
descending (HOL method), **294**
descending_setup (HOL method), **294**
description (document marker), **85**
dest (attribute), **244**
dest (Pure attribute), **153**
display (antiquotation option), **82**
document_tags (system option), 86
done (command), **170**
drule (method), **214**
drule_tac (method), **175**

elim (attribute), **244**
elim (method), **214**
elim (Pure attribute), **153**
elim_format (Pure attribute), **216**
elim_resolution (inference), **32**
embedded (syntax), **57**
emph (antiquotation), **74**
end (global command), **92**
end (local command), **95**, 117
erule (method), **214**
erule_tac (method), **175**
eta_contract (antiquotation option), **82**
eta_contract (attribute), **181**, 207
expand (inference), **34**
experiment (command), **106**
export (inference), **34**
export (tool), 323
export_code (command), 126
export_code (HOL command), **313**
export_generated_files (command), **126**
external_file (command), **126**

fact (method), 64, **153**
fail (method), **214**
fast (method), **246**
fastforce (method), **246**
file (antiquotation), **74**
finally (command), **146**
find_consts (command), **67**
find_theorems (command), **67**
find_unused_assms (HOL command), **300**
finish (inference), **31**
fix (command), **137**
fixes (element), 106
float (syntax), **54**
float_const (inner syntax), **192**
float_token (inner syntax), **191**
fold (method), **214**
folded (attribute), **216**
for (keyword), **132**
for_fixes (syntax), **66**
force (method), **246**
from (command), **140**
frule (method), **214**
frule_tac (method), **175**
full_prf (antiquotation), **73**
full_prf (command), **178**
fun (HOL command), **264**
fun_cases (HOL command), **264**
function (HOL command), **264**
functor (HOL command), **283**

generate_file (command), **126**
global_interpretation (command), **110**
goal_cases (method), **153**
goal_spec (syntax), **150**
goals (antiquotation), **73**

goals_limit (antiquotation option), **83**
goals_limit (attribute), **181**

have (command), **142**
help (command), **52**
hence (command), **142**
hide_class (command), **133**
hide_const (command), **133**
hide_fact (command), **133**
hide_type (command), **133**
hypsubst (method), **217**

id (inner syntax), **191**
idt (inner syntax), **194**, 195, 196
idts (inner syntax), **194**, 196
if (keyword), 146, 169
iff (attribute), **244**
in (keyword), **96**
include (command), **97**
includes (keyword), **97**
includes (syntax), 96, **98**, 144
including (command), **97**
ind_cases (HOL method), **310**
indent (antiquotation option), **83**
index (inner syntax), **194**, 195
induct (attribute), **165**
induct (method), 142, 159, **160**
induct_simp (attribute), **164**
induct_tac (HOL method), **310**
induction (method), **160**
induction_schema (HOL method), **269**
inductive (HOL command), **260**
inductive_cases (HOL command), **310**
inductive_set (HOL command), **260**
infix (keyword), **188**
infix (notation kind), **188**
infixl (keyword), **188**
infixr (keyword), **188**
init (inference), **31**
injection (HOL method), **294**
insert (method), **214**
inst (syntax), **59**
inst_step (method), **251**
instance (command), **115**, 133
instantiation (command), **115**, 133
insts (syntax), **60**
int (syntax), **56**
interpret (command), **110**
intro (attribute), **244**
intro (method), **214**
intro (Pure attribute), **153**
intro_classes (method), **115**, 152
intro_locales (method), **110**, 152
iprover (HOL method), **306**
is (keyword), **139**

judgment (command), **253**
judgment (notation kind), **188**

keywords (keyword), **94**

lemma (antiquotation), **73**
lemma (command), **142**
lemmas (command), **132**
let (command), **139**
lexicographic_order (HOL method), **269**
license (document marker), **85**
lift_definition (HOL command), **286**, **288–290**
lifting (HOL method), **294**
lifting_forget (HOL command), **286**
lifting_restore (HOL attribute), **286**
lifting_setup (HOL method), **294**
lifting_update (HOL command), **286**
linarith_split (HOL attribute), **306**
literal (notation kind), **188**
local_setup (command), **122**
locale (antiquotation), **73**

locale (command), **106**
locale (syntax), **107**
locale_deps (command), **106**
locale_expr (syntax), **104**
logic (inner syntax), **194**, 195
long_ident (syntax), **54**, 191
longid (inner syntax), **191**

margin (antiquotation option), **83**
marker (syntax), **84**
marker_body (syntax), **85**
meson (HOL method), **307**
method (syntax), **149**
method_facts (fact), **140**
method_setup (command), **156**
metis (HOL method), **307**
mixfix (notation kind), **188**
mixfix (syntax), **184**
mixfix_properties (syntax), 186, **187**
mkroot (tool), 71
ML (antiquotation), **73**
ML (command), **122**
ML_command (command), **122**
ML_debugger (attribute), **122**
ML_debugger (system option), 125
ML_def (antiquotation), **73**
ML_environment (attribute), **122**
ML_exception_debugger (attribute), **122**
ML_exception_trace (attribute), **122**
ML_export (command), **122**
ML_file (command), **122**
ML_file_debug (command), **122**
ML_file_no_debug (command), **122**
ML_functor (antiquotation), **73**
ML_functor_def (antiquotation), **73**
ML_functor_ref (antiquotation), **73**
ML_infix (antiquotation), **73**
ML_infix_def (antiquotation), **73**
ML_infix_ref (antiquotation), **73**
ML_prf (command), **122**
ML_print_depth (attribute), **122**
ML_ref (antiquotation), **73**
ML_source_trace (attribute), **122**
ML_structure (antiquotation), **73**
ML_structure_def (antiquotation), **73**
ML_structure_ref (antiquotation), **73**
ML_text (antiquotation), **73**
ML_type (antiquotation), **73**
ML_type_def (antiquotation), **73**
ML_type_ref (antiquotation), **73**
ML_val (command), **122**
mode (antiquotation option), **82**
modes (syntax), **179**
mono (HOL attribute), **260**
moreover (command), **146**
multi_specs (syntax), **66**

name (syntax), **55**, 75
named_inst (syntax), **60**
named_insts (syntax), **60**
named_theorems (command), **132**
names_long (antiquotation option), **82**
names_long (attribute), **181**
names_short (antiquotation option), **82**
names_short (attribute), **181**
names_unique (antiquotation option), **82**
names_unique (attribute), **181**
nat (syntax), **54**, 54, 191
next (command), **136**
nitpick (HOL command), **300**
nitpick_params (HOL command), **300**
no_adhoc_overloading (command), **121**

no_notation (command), **190**
no_syntax (command), **202**
no_translations (command), **202**
no_type_notation (command), **190**
no_vars (attribute), **216**
nocite (antiquotation), **74**
nonterminal (command), **202**
notation (command), **190**
note (command), **140**
notepad (command), **135**
notes (element), 106
nothing (fact), 142
num_const (inner syntax), **192**
num_token (inner syntax), **191**

obtain (command), **166**
obtain_case (syntax), **145**
obtain_clauses (syntax), **144**
obtains (element), **143**
OF (attribute), **153**
of (attribute), **153**
old_rep_datatype (HOL command), **274**
oops (command), **137**
opening (keyword), **97**
opening (syntax), 96, **98**, 107, 115
options (keyword), **94**
oracle (command), **132**
output (keyword), 204
overloaded (syntax), **281**
overloading (command), **119**

par_name (syntax), **56**
paragraph (command), **71**
params (attribute), **157**
parse_ast_translation (command), **207**
parse_translation (command), **207**
partial_function (HOL command), **271**
partial_function_mono (HOL attribute), **271**
partiality_descending (HOL method), **294**
partiality_descending_setup (HOL method), **294**
pat_completeness (HOL method), **269**
postfix (notation kind), **188**
prefer (command), **170**
prefix (notation kind), **188**
presume (command), **137**
prf (antiquotation), **73**
prf (command), **178**
primrec (HOL command), **264**
print_abbrevs (command), **100**
print_antiquotations (command), **74**
print_ast_translation (command), **207**
print_attributes (command), **67**
print_bundles (command), **97**
print_cases (command), **157**
print_claset (command), **244**
print_classes (command), **115**
print_codeproc (HOL command), **313**
print_codesetup (HOL command), **313**
print_commands (command), **52**
print_definitions (command), **67**
print_defn_rules (command), **100**
print_facts (command), **67**
print_induct_rules (command), **165**
print_inductives (command), **260**
print_interps (command), **110**
print_locale (command), **106**
print_locales (command), **106**
print_methods (command), **67**
print_options (command), **213**
print_quot_maps (HOL command), **286**
print_quotconsts (HOL command), **294**

print_quotients (HOL command), **286**
print_quotientsQ3 (HOL command), **294**
print_quotmapsQ3 (HOL command), **294**
print_record (HOL command), **277**
print_rules (command), **153**
print_simpset (command), **224**
print_state (command), **178**
print_statement (command), **142**
print_syntax (command), **197**, 207, 208
print_term_bindings (command), **67**
print_theorems (command), **67**
print_theory (command), **67**
print_trans_rules (command), **146**
print_translation (command), **207**
`Print_Mode.with_modes` (ML), **183**
`print_mode_value` (ML), **183**
private (keyword), **95**
proof
  fake, 152
  standard, 152
  terminal, 152
  trivial, 152
proof (command), 141, 142, **150**, 150, 154
prop (antiquotation), **73**
prop (command), **178**
prop (inner syntax), **193**, 195
prop (syntax), **59**
prop_pat (syntax), **62**
proposition (command), **142**
props (syntax), **62**
props' (syntax), **63**
pttrn (inner syntax), **194**, 196
pttrns (inner syntax), **194**, 196

qed (command), **150**, 150
qualified (keyword), **95**
quickcheck (HOL command), **300**
quickcheck_generator (HOL command), **300**
quickcheck_params (HOL command), **300**
quot_del (HOL attribute), **286**
quot_lifted (HOL attribute), **294**
quot_map (HOL attribute), **286**
quot_preserve (HOL attribute), **294**
quot_respect (HOL attribute), **294**
quot_thm (HOL attribute), **294**
quotes (antiquotation option), **82**
quotient_definition (HOL command), **294**
quotient_type (HOL command), **284**

rail (antiquotation), **87**
raw_tactic (method), **175**
real (syntax), **56**
recdef (HOL command), **272**
recdef_cong (HOL attribute), **273**
recdef_simp (HOL attribute), **273**
recdef_wf (HOL attribute), **273**
record (HOL command), **277**
regularize (HOL method), **294**
relation (HOL method), **269**
relator_distr (HOL attribute), **286**
relator_domain (HOL attribute), **292, 293**
relator_eq (HOL attribute), **292**
relator_eq_onp (HOL attribute), **286**
relator_mono (HOL attribute), **286**
rename_tac (method), **175**
resolution (inference), **31**
rotate_tac (method), **175**
rotated (attribute), **216**
rule (attribute), **244**
rule (HOL method), 151
rule (method), **245**

rule (Pure attribute), **153**
rule (Pure method), 142, 151, 152, **153**, 154
rule_format (attribute), **253**
rule_tac (method), **175**
rulify (attribute), **253**

safe (method), **250**
safe_step (method), **251**
schematic_goal (command), **142**
section (command), **71**
session (antiquotation), **74**
setup (command), **122**
setup_lifting (HOL command), **286**
short_ident (syntax), **54**, 191
show (command), 138, **142**, 150
show_abbrevs (antiquotation option), **82**
show_abbrevs (attribute), **181**
show_consts (attribute), **181**
show_consts_markup (attribute), **181**
show_hyps (attribute), **181**
show_main_goal (attribute), **181**
show_markup (attribute), **181**
show_question_marks (attribute), **181**
show_sorts (antiquotation option), **82**
show_sorts (attribute), **181**
show_structs (antiquotation option), **82**
show_tags (attribute), **181**
show_types (antiquotation option), **82**
show_types (attribute), **181**
show_variants (attribute), **121**
shows (element), **143**
simp (attribute), **224**
simp (method), **219**
simp (Pure method), **219**
simp_all (method), **219**
simp_all (Pure method), **219**
simp_break (attribute), **229**
simp_debug (attribute), **229**
simp_depth_limit (attribute), **219**
simp_trace (attribute), **229**
simp_trace_depth_limit (attribute), **229**
simp_trace_new (attribute), **229**
simplified (attribute), **238**
`Simplifier.add_loop` (ML), **237**
`Simplifier.add_safe_solver` (ML), **235**
`Simplifier.add_unsafe_solver` (ML), **235**
`Simplifier.del_loop` (ML), **237**
`Simplifier.mk_solver` (ML), **235**
`Simplifier.prems_of` (ML), **234**
`Simplifier.set_loop` (ML), **237**
`Simplifier.set_safe_solver` (ML), **235**
`Simplifier.set_subgoaler` (ML), **234**
`Simplifier.set_term_ord` (ML), **228**
`Simplifier.set_unsafe_solver` (ML), **235**
simpmod (syntax), **220**
simproc_setup (command), **231**
simproc_setup (ML antiquotation), **231**, 233
simproc_setup (syntax), **231**
simproc_setup_id (syntax), **232**
size_change (HOL method), **269**
sledgehammer (HOL command), **297**
sledgehammer_params (HOL command), **297**
sleep (method), **214**
slow (method), **246**
slow_step (method), **251**
slowsimp (method), **246**

SML_file (command), **122**
SML_file_debug (command), **122**
SML_file_no_debug (command), **122**
solve_direct (HOL command), **297**
`solver` (ML type), **235**
sorry (command), 137, **150**
sort (inner syntax), **195**, 196
sort (syntax), **58**
source (antiquotation option), **83**
source_cartouche (antiquotation option), **83**
spec_prems (syntax), **67**
specification (HOL command), **273**
specification (syntax), **67**
split (attribute), **224**
split (method), **217**, 221
split_format (HOL attribute), **311**
`Splitter.add_split` (ML), **237**
`Splitter.add_split_bang` (ML), **237**
`Splitter.del_split` (ML), **237**
standard (method), **150**
step (method), **251**
str_token (inner syntax), **191**
string (syntax), **54**, 54
string_token (inner syntax), **191**
structure (keyword), 109, 195
structured_spec (syntax), **66**
subclass (command), **115**
subgoal (command), **172**
subgoal_tac (method), **175**
subgoals (antiquotation), **73**
sublocale (command), **110**
subparagraph (command), **71**
subproofs (method), **153**
subsection (command), **71**
subst (method), **217**
subsubsection (command), **71**
succeed (method), **214**
supply (command), **170**
swapped (attribute), **244**
sym (attribute), **146**
sym_ident (syntax), **54**
symmetric (attribute), **146**
syntax (command), **202**
syntax_ambiguity_limit (attribute), **198**
syntax_ambiguity_warning (attribute), **198**
syntax_ast_stats (attribute), **202**
syntax_ast_trace (attribute), **202**
syntax_const (ML antiquotation), **207**
syntax_consts (command), **202**
syntax_declaration (command), **103**
syntax_types (command), **202**
system_name (syntax), **56**
system_option (antiquotation), **74**

tactic (method), **175**
tag (document marker), **86**
tagged (attribute), **216**
tags (syntax), **85**
target (syntax), **96**
term (antiquotation), **73**
term (command), **178**
term (syntax), **59**
term abbreviations, 140
term_pat (syntax), **62**
term_type (antiquotation), **73**
term_var (syntax), **54**, 54
termination (HOL command), **264**
termination_simp (HOL attribute), **269**
text (antiquotation), **73**, 74
text (command), **71**, 83
text (syntax), **57**
text_raw (command), **71**, 83
that (fact), 146, 169
THEN (attribute), **216**
then (command), **140**

theorem (command), **142**
theory (antiquotation), **73**
theory (command), **92**
thesis (variable), 140
thin_tac (method), **175**
this (fact), 135, 140
this (method), **153**
this (variable), 140
thm (antiquotation), **73**
thm (command), **178**
thm (syntax), **65**
thm_deps (command), **67**
thm_oracles (command), **132**
thmbind (syntax), **65**
thmdecl (syntax), **65**
thmdef (syntax), **65**
thms (syntax), **65**
thus (command), **142**
thy_deps (command), **92**
tid (inner syntax), **191**
title (document marker), **85**
trace_locales (attribute), **110**
trans (attribute), **146**
transfer (HOL method), **292**
transfer' (HOL method), **292**
Transfer.transferred (HOL attribute), **292**
transfer_domain_rule (HOL attribute), **292**
transfer_end (HOL method), **292**
transfer_prover (HOL method), **292**
transfer_prover_end (HOL method), **292**
transfer_prover_start (HOL method), **292**
transfer_rule (HOL attribute), **292**
transfer_start (HOL method), **292**
transfer_step (HOL method), **292**
translations (command), **202**
try (HOL command), **297**
try0 (HOL command), **297**
tvar (inner syntax), **191**
txt (command), **71**, 83
typ (antiquotation), **73**
typ (command), **178**
type (antiquotation), **73**
type (inner syntax), **194**, 196
type (syntax), **59**
type_alias (command), **133**
type_application (notation kind), **188**
type_ident (syntax), **54**, 191
type_name (inner syntax), **195**
type_notation (command), **190**
type_synonym (command), **131**
type_syntax (ML antiquotation), **207**
type_var (syntax), **54**, 54, 191
typeargs (syntax), **61**
typeargs_sorts (syntax), **61**
typed_print_translation (command), **207**
typedecl (command), **131**, 131
typedef (command), **131**
typedef (HOL command), **280**
typeof (antiquotation), **73**
typespec (syntax), **61**
typespec_sorts (syntax), **61**

ultimately (command), **146**
unbundle (command), **97**
unfold (method), 141, **214**
unfold_locales (method), **110**
unfolded (attribute), **216**
unfolding (command), **140**
unify_search_bound (attribute), **255**
unify_trace (attribute), **255**
unify_trace_bound (attribute), **255**
unify_trace_simp (attribute), **255**
unify_trace_types (attribute), **255**
untagged (attribute), **216**

untransferred (HOL attribute), **292**
unused_thms (command), **67**
url (antiquotation), **74**
use (method), **140**
using (command), **140**

value (HOL command), 78, **300**
values (HOL command), **300**
var (inner syntax), **191**
var (syntax), 191
vars (syntax), **62**
verbatim (antiquotation), **74**
verbatim (syntax), **54**, 54

when (keyword), 146
where (attribute), **153**
with (command), **140**
`wrapper` (ML type), **252**
write (command), **190**