Isabelle’s Logics

Lawrence C. Paulson
Computer Laboratory
University of Cambridge
lcp@cl.cam.ac.uk

With Contributions by Tobias Nipkow and Markus Wenzel

15 April 2020

1Markus Wenzel made numerous improvements. Sara Kalvala contributed Chap. 4. Philippe de Groote wrote the first version of the logic LK. Tobias Nipkow developed LCF and Cube. Martin Coen developed Modal with assistance from Rajeev Goré. The research has been funded by the EPSRC (grants GR/G53279, GR/H40570, GR/K57381, GR/K77051, GR/M75440) and by ESPRIT (projects 3245: Logical Frameworks, and 6453: Types), and by the DFG Schwerpunktprogramm Deduktion.
Contents

1 Syntax definitions

2 Higher-Order Logic
   2.1 Syntax
      2.1.1 Types and overloading
      2.1.2 Binders
      2.1.3 The let and case constructions
   2.2 Rules of inference
   2.3 A formulation of set theory
      2.3.1 Syntax of set theory
      2.3.2 Axioms and rules of set theory
      2.3.3 Properties of functions
   2.4 Simplification and substitution
      2.4.1 Case splitting
   2.5 Types
      2.5.1 Product and sum types
      2.5.2 The type of natural numbers, nat
      2.5.3 Numerical types and numerical reasoning
      2.5.4 The type constructor for lists, list
   2.6 Datatype definitions
      2.6.1 Basics
      2.6.2 Defining datatypes
   2.7 Old-style recursive function definitions
   2.8 Example: Cantor’s Theorem

3 First-Order Sequent Calculus
   3.1 Syntax and rules of inference
   3.2 Automatic Proof
   3.3 Tactics for the cut rule
   3.4 Tactics for sequents
   3.5 A simple example of classical reasoning
   3.6 A more complex proof
   3.7 *Unification for lists
   3.8 *Packaging sequent rules
3.9 *Proof procedures ................................................. 57
  3.9.1 Method A ................................................. 57
  3.9.2 Method B ................................................. 58

4 Defining A Sequent-Based Logic 59
  4.1 Concrete syntax of sequences ................................. 59
  4.2 Basis .......................................................... 60
  4.3 Object logics .................................................. 60
  4.4 What’s in Sequents.thy ....................................... 61

5 Constructive Type Theory 63
  5.1 Syntax .......................................................... 65
  5.2 Rules of inference .............................................. 69
  5.3 Rule lists ....................................................... 72
  5.4 Tactics for subgoal reordering ............................... 73
  5.5 Rewriting tactics ............................................... 73
  5.6 Tactics for logical reasoning ................................ 74
  5.7 A theory of arithmetic ...................................... 75
  5.8 The examples directory ...................................... 75
  5.9 Example: type inference ...................................... 77
  5.10 An example of logical reasoning ............................ 78
  5.11 Example: deriving a currying functional ................. 81
  5.12 Example: proving the Axiom of Choice .................... 83
Several logics come with Isabelle. Many of them are sufficiently developed to serve as comfortable reasoning environments. They are also good starting points for defining new logics. Each logic is distributed with sample proofs, some of which are described in this document.

**HOL** is currently the best developed Isabelle object-logic, including an extensive library of (concrete) mathematics, and various packages for advanced definitional concepts (like (co-)inductive sets and types, well-founded recursion etc.). The distribution also includes some large applications.

**ZF** provides another starting point for applications, with a slightly less developed library than **HOL**. **ZF**’s definitional packages are similar to those of **HOL**. Untyped **ZF** set theory provides more advanced constructions for sets than simply-typed **HOL**. **ZF** is built on **FOL** (first-order logic), both are described in a separate manual *Isabelle’s Logics: FOL and ZF* [12].

There are some further logics distributed with Isabelle:

**CCL** is Martin Coen’s Classical Computational Logic, which is the basis of a preliminary method for deriving programs from proofs [2]. It is built upon classical **FOL**.

**LCF** is a version of Scott’s Logic for Computable Functions, which is also implemented by the **lcf** system [13]. It is built upon classical **FOL**.

**HOLCF** is a version of **LCF**, defined as an extension of **HOL**. See [10] for more details on **HOLCF**.

**CTT** is a version of Martin-Löf’s Constructive Type Theory [11], with extensional equality. Universes are not included.

**Cube** is Barendregt’s λ-cube.

The directory **Sequents** contains several logics based upon the sequent calculus. Sequents have the form $A_1, \ldots, A_m \vdash B_1, \ldots, B_n$; rules are applied using associative matching.

**LK** is classical first-order logic as a sequent calculus.

**Modal** implements the modal logics $T$, $S4$, and $S43$. 

---

1
ILL implements intuitionistic linear logic.

The logics CCL, LCF, Modal, ILL and Cube are undocumented. All object-logics’ sources are distributed with Isabelle (see the directory src). They are also available for browsing on the WWW at

http://www.cl.cam.ac.uk/Research/HVG/Isabelle/library/
https://isabelle.in.tum.de/library/

Note that this is not necessarily consistent with your local sources!

Do not read the Isabelle’s Logics manuals before reading Isabelle/HOL — The Tutorial or Introduction to Isabelle, and performing some Isabelle proofs. Consult the Reference Manual for more information on tactics, packages, etc.
Chapter 1

Syntax definitions

The syntax of each logic is presented using a context-free grammar. These grammars obey the following conventions:

- identifiers denote nonterminal symbols
- typewriter font denotes terminal symbols
- parentheses (…) express grouping
- constructs followed by a Kleene star, such as id∗ and (…)∗ can be repeated 0 or more times
- alternatives are separated by a vertical bar, |
- the symbol for alphanumeric identifiers is id
- the symbol for scheme variables is var

To reduce the number of nonterminals and grammar rules required, Isabelle’s syntax module employs priorities, or precedences. Each grammar rule is given by a mixfix declaration, which has a priority, and each argument place has a priority. This general approach handles infix operators that associate either to the left or to the right, as well as prefix and binding operators.

In a syntactically valid expression, an operator’s arguments never involve an operator of lower priority unless brackets are used. Consider first-order logic, where ∃ has lower priority than ∨, which has lower priority than ∧. There, P ∧ Q ∨ R abbreviates (P ∧ Q) ∨ R rather than P ∧ (Q ∨ R). Also, ∃x . P ∨ Q abbreviates ∃x . (P ∨ Q) rather than (∃x . P) ∨ Q. Note especially that P ∨ (∃x . Q) becomes syntactically invalid if the brackets are removed.

A binder is a symbol associated with a constant of type (σ ⇒ τ) ⇒ τ’. For instance, we may declare ∀ as a binder for the constant All, which has type (α ⇒ o) ⇒ o. This defines the syntax ∀x . t to mean All(λx . t). We can also write ∀x1 . . . xn . t to abbreviate ∀x1 . . . ∀xn . t; this is possible for any constant provided that τ and τ’ are the same type. The Hilbert description operator εx . P x has type (α ⇒ bool) ⇒ α and normally binds only one
variable. ZF’s bounded quantifier \( \forall x \in A . P(x) \) cannot be declared as a binder because it has type \([i, i \Rightarrow o] \Rightarrow o\). The syntax for binders allows type constraints on bound variables, as in

\[
\forall(x::\alpha) \ (y::\beta) \ z::\gamma . \ Q(x, y, z)
\]

To avoid excess detail, the logic descriptions adopt a semi-formal style. Infix operators and binding operators are listed in separate tables, which include their priorities. Grammar descriptions do not include numeric priorities; instead, the rules appear in order of decreasing priority. This should suffice for most purposes; for full details, please consult the actual syntax definitions in the .thy files.

Each nonterminal symbol is associated with some Isabelle type. For example, the formulae of first-order logic have type \( o \). Every Isabelle expression of type \( o \) is therefore a formula. These include atomic formulae such as \( P \), where \( P \) is a variable of type \( o \), and more generally expressions such as \( P(t, u) \), where \( P \), \( t \) and \( u \) have suitable types. Therefore, ‘expression of type \( o \)’ is listed as a separate possibility in the grammar for formulae.
Chapter 2

Higher-Order Logic

This chapter describes Isabelle’s formalization of 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 monograph *Isabelle/HOL — A Proof Assistant for Higher-Order Logic* provides a gentle introduction on using Isabelle/HOL in practice. All of this material is mainly of historical interest!

2.1 Syntax

Figure 2.1 lists the constants (including infixes and binders), while Fig. 2.2 presents the grammar of higher-order logic. Note that $a \neq b$ is translated to $\neg (a = b)$.

HOL has no if-and-only-if connective; logical equivalence is expressed using equality. But equality has a high priority, as befitting a relation, while if-and-only-if typically has the lowest priority. Thus, $\neg \neg P = P$ abbreviates $\neg \neg (P = P)$ and not $(\neg \neg P) = P$. When using $=$ to mean logical equivalence, enclose both operands in parentheses.

2.1.1 Types and overloading

The universal type class of higher-order terms is called *term*. By default, explicit type variables have class *term*. In particular the equality symbol and quantifiers are polymorphic over class *term*.

The type of formulae, *bool*, belongs to class *term*; thus, formulae are terms. The built-in type *fun*, which constructs function types, is overloaded with arity (*term*, *term*) *term*. Thus, $\sigma \Rightarrow \tau$ belongs to class *term* if $\sigma$ and $\tau$ do, allowing quantification over functions.

HOL allows new types to be declared as subsets of existing types, either using the primitive *typedef* or the more convenient *datatype* (see §2.6).

Several syntactic type classes — *plus*, *minus*, *times* and *power* — permit overloading of the operators $+$, $-$, $\ast$, and $^\wedge$. They are overloaded to denote
### CHAPTER 2. HIGHER-ORDER LOGIC

<table>
<thead>
<tr>
<th>name</th>
<th>meta-type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trueprop</td>
<td>bool ⇒ prop</td>
<td>coercion to prop</td>
</tr>
<tr>
<td>Not</td>
<td>bool ⇒ bool</td>
<td>negation (¬)</td>
</tr>
<tr>
<td>True</td>
<td>bool</td>
<td>tautology (⊤)</td>
</tr>
<tr>
<td>False</td>
<td>bool</td>
<td>absurdity (⊥)</td>
</tr>
<tr>
<td>If</td>
<td>[bool, α, α] ⇒ α</td>
<td>conditional</td>
</tr>
<tr>
<td>Let</td>
<td>[α, α ⇒ β] ⇒ β</td>
<td>let binder</td>
</tr>
</tbody>
</table>

### Constants

<table>
<thead>
<tr>
<th>symbol</th>
<th>name</th>
<th>meta-type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOME or ⊥</td>
<td>Eps</td>
<td>(α ⇒ bool) ⇒ α</td>
<td>Hilbert description (ε)</td>
</tr>
<tr>
<td>ALL or !</td>
<td>All</td>
<td>(α ⇒ bool) ⇒ bool</td>
<td>universal quantifier (∀)</td>
</tr>
<tr>
<td>EX or ?</td>
<td>Ex</td>
<td>(α ⇒ bool) ⇒ bool</td>
<td>existential quantifier (∃)</td>
</tr>
<tr>
<td>EX! or ?!</td>
<td>Ex1</td>
<td>(α ⇒ bool) ⇒ bool</td>
<td>unique existence (∃!)</td>
</tr>
<tr>
<td>LEAST</td>
<td>Least</td>
<td>(α :: ord ⇒ bool) ⇒ α</td>
<td>least element</td>
</tr>
</tbody>
</table>

### Binders

<table>
<thead>
<tr>
<th>symbol</th>
<th>meta-type</th>
<th>priority</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>o</td>
<td>[β ⇒ γ, α ⇒ β] ⇒ (α ⇒ γ)</td>
<td>Left 55</td>
<td>composition (◦)</td>
</tr>
<tr>
<td>=</td>
<td>[α, α] ⇒ bool</td>
<td>Left 50</td>
<td>equality (=)</td>
</tr>
<tr>
<td>&lt;</td>
<td>[α :: ord, α] ⇒ bool</td>
<td>Left 50</td>
<td>less than (&lt;)</td>
</tr>
<tr>
<td>&lt;=</td>
<td>[α :: ord, α] ⇒ bool</td>
<td>Left 50</td>
<td>less than or equals (≤)</td>
</tr>
<tr>
<td>&amp;</td>
<td>[bool, bool] ⇒ bool</td>
<td>Right 35</td>
<td>conjunction (∧)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[bool, bool] ⇒ bool</td>
<td>Right 30</td>
</tr>
<tr>
<td>--&gt;</td>
<td>[bool, bool] ⇒ bool</td>
<td>Right 25</td>
<td>implication (→)</td>
</tr>
</tbody>
</table>

### Infixes

Figure 2.1: Syntax of HOL
CHAPTER 2. HIGHER-ORDER LOGIC

\[ term = \text{expression of class term} \]
\[ \ | \ \text{SOME } id . \ \text{formula} \ | \ \@ id . \ \text{formula} \]
\[ \text{let } id = \text{term}; \ldots ; \ \text{id = term in term} \]
\[ \text{if } \text{formula then } \text{term else } \text{term} \]
\[ \text{LEAST } id . \ \text{formula} \]

\[ \text{formula } = \text{expression of type bool} \]
\[ \text{term } = \text{term} \]
\[ \text{term } \text{=} \text{term} \]
\[ \text{term } < \text{term} \]
\[ \text{term } \leq \text{term} \]
\[ \sim \ \text{formula} \]
\[ \text{formula } \& \ \text{formula} \]
\[ \text{formula } | \ \text{formula} \]
\[ \text{formula } \rightarrow \ \text{formula} \]
\[ \text{ALL } id id^* . \ \text{formula} \ | \ ! \ id id^* . \ \text{formula} \]
\[ \text{EX } id id^* . \ \text{formula} \ | \ ? \ id id^* . \ \text{formula} \]
\[ \text{EX! } id id^* . \ \text{formula} \ | \ ?! \ id id^* . \ \text{formula} \]

Figure 2.2: Full grammar for HOL
the obvious arithmetic operations on types \texttt{nat}, \texttt{int} and \texttt{real}. (With the \(^\wedge\) operator, the exponent always has type \texttt{nat}.) Non-arithmetic overloading
are also done: the operator - can denote set difference, while \(^\wedge\) can denote
exponentiation of relations (iterated composition). Unary minus is also written as - and is overloaded like its 2-place counterpart; it even can stand for
set complement.

The constant \texttt{0} is also overloaded. It serves as the zero element of several
types, of which the most important is \texttt{nat} (the natural numbers). The type
class \texttt{plus_ac0} comprises all types for which \texttt{0} and + satisfy the laws \(x + y = y + x, (x + y) + z = x + (y + z)\) and \(0 + x = x\). These types include the numeric
ones \texttt{nat}, \texttt{int} and \texttt{real} and also multisets. The summation operator \texttt{sum}
is available for all types in this class.

Theory \texttt{Ord} defines the syntactic class \texttt{ord} of order signatures. The relations \(<\) and \(\leq\) are polymorphic over this class, as are the functions \texttt{mono}, \texttt{min}
and \texttt{max}, and the \texttt{LEAST} operator. \texttt{Ord} also defines a subclass \texttt{order} of \texttt{ord}
which axiomatizes the types that are partially ordered with respect to \(\leq\). A
further subclass \texttt{linorder} of \texttt{order} axiomatizes linear orderings. For details,
see the file \texttt{Ord.thy}.

If you state a goal containing overloaded functions, you may need to
include type constraints. Type inference may otherwise make the goal more
polymorphic than you intended, with confusing results. For example, the
variables \(i, j, k\) in the goal \(i \leq j \Rightarrow i \leq j + k\) have type \(\alpha :: \{\texttt{ord}, \texttt{plus}\}\),
although you may have expected them to have some numeric type, e.g. \texttt{nat}.
Instead you should have stated the goal as \((i :: \texttt{nat}) \leq j \Rightarrow i \leq j + k\), which
causes all three variables to have type \texttt{nat}.

If resolution fails for no obvious reason, try setting \texttt{show_types} to \texttt{true}, caus-
ing Isabelle to display types of terms. Possibly set \texttt{show_sorts} to \texttt{true} as well,
causing Isabelle to display type classes and sorts.

Where function types are involved, Isabelle’s unification code does not guar-
antee to find instantiations for type variables automatically. Be prepared to use
\texttt{res_inst_tac} instead of \texttt{resolve_tac}, possibly instantiating type variables. Set-
ting \texttt{Unify.trace_types} to \texttt{true} causes Isabelle to report omitted search paths
during unification.

2.1.2 Binders

Hilbert’s \texttt{description} operator \(\varepsilon x. P[x]\) stands for some \(x\) satisfying \(P\), if
such exists. Since all terms in HOL denote something, a description is always
meaningful, but we do not know its value unless \(P\) defines it uniquely. We
may write descriptions as \texttt{Eps}(\(\lambda x. P[x]\)) or use the syntax \texttt{SOME} \(x\). \(P[x]\).
Existential quantification is defined by
\[ \exists x . \, P \, x \equiv P(\varepsilon \, x . \, P \, x) . \]
The unique existence quantifier, \( \exists! x . \, P \), is defined in terms of \( \exists \) and \( \forall \). An Isabelle binder, it admits nested quantifications. For instance, \( \exists! x \, y . \, P \, x \, y \) abbreviates \( \exists x . \, \exists y . \, P \, x \, y \); note that this does not mean that there exists a unique pair \((x, y)\) satisfying \( P \, x \, y \).

The basic Isabelle/HOL binders have two notations. Apart from the usual \texttt{ALL} and \texttt{EX} for \( \forall \) and \( \exists \), Isabelle/HOL also supports the original notation of Gordon’s HOL system: \( ! \) and \( ? \). In the latter case, the existential quantifier must be followed by a space; thus \( ? x \) is an unknown, while \( ? x . \, f \, x = y \) is a quantification. Both notations are accepted for input. The print mode “HOL” governs the output notation. If enabled (e.g. by passing option \(-m\) \texttt{HOL} to the \texttt{isabelle} executable), then \( ! \) and \( ? \) are displayed.

If \( \tau \) is a type of class \texttt{ord}, \( P \) a formula and \( x \) a variable of type \( \tau \), then the term \( \text{LEAST} \, x . \, P \, [\, x \,] \) is defined to be the least (w.r.t. \( \leq \)) \( x \) such that \( P \, x \) holds (see Fig. 2.4). The definition uses Hilbert’s \( \varepsilon \) choice operator, so \texttt{Least} is always meaningful, but may yield nothing useful in case there is not a unique least element satisfying \( P \).

All these binders have priority 10.

The low priority of binders means that they need to be enclosed in parenthesis when they occur in the context of other operations. For example, instead of \( P \land \forall x . \, Q \) you need to write \( P \land (\forall x . \, Q) \).

### 2.1.3 The let and case constructions

Local abbreviations can be introduced by a \texttt{let} construct whose syntax appears in Fig. 2.2. Internally it is translated into the constant \texttt{Let}. It can be expanded by rewriting with its definition, \texttt{Let_def}.

HOL also defines the basic syntax
\[
\text{case } e \ \text{of} \ c_1 \Rightarrow e_1 \mid \ldots \mid c_n \Rightarrow e_n
\]
as a uniform means of expressing \texttt{case} constructs. Therefore \texttt{case} and \texttt{of} are reserved words. Initially, this is mere syntax and has no logical meaning. By declaring translations, you can cause instances of the \texttt{case} construct to denote applications of particular case operators. This is what happens automatically for each \texttt{datatype} definition (see §2.6).

\(^1\)Class \texttt{ord} does not require much of its instances, so \( \leq \) need not be a well-ordering, not even an order at all!
Both if and case constructs have as low a priority as quantifiers, which requires additional enclosing parentheses in the context of most other operations. For example, instead of \( f \ x = \text{if} \ldots \text{then} \ldots \text{else} \ldots \) you need to write \( f \ x = (\text{if} \ldots \text{then} \ldots \text{else} \ldots) \).

## 2.2 Rules of inference

Figure 2.3 shows the primitive inference rules of HOL, with their ML names. Some of the rules deserve additional comments:

- **ext** expresses extensionality of functions.
- **iff** asserts that logically equivalent formulae are equal.
- **someI** gives the defining property of the Hilbert \( \varepsilon \)-operator. It is a form of the Axiom of Choice. The derived rule **some_equality** (see below) is often easier to use.
- **True_or_False** makes the logic classical.\(^2\)

HOL follows standard practice in higher-order logic: only a few connectives are taken as primitive, with the remainder defined obscurely (Fig. 2.4). Gordon’s HOL system expresses the corresponding definitions [6, page 270] using object-equality (=), which is possible because equality in higher-order logic may equate formulae and even functions over formulae. But theory HOL, like all other Isabelle theories, uses meta-equality (\(\equiv\)) for definitions.

---

\(^2\)In fact, the \( \varepsilon \)-operator already makes the logic classical, as shown by Diaconescu; see Paulson [14] for details.
CHAPTER 2. HIGHER-ORDER LOGIC

True_def True == (%(x::bool. x)=(%x. x))
All_def All == (%P. P = (%x. True))
Ex_def Ex == (%P. P(@x. P x))
False_def False == (!P. P)
not_def not == (%P. P-->False)
and_def op & == (%P Q. !R. (P-->Q-->R) --> R)
or_def op | == (%P Q. !R. (P-->R) --> (Q-->R) --> R)
Ex1_def Ex1 == (%P. ? x. P x & (! y. P y --> y=x))
o_def op o == (%(f::'b=>'c) g x::'a. f(g x))
if_def If P x y == (%P x y. @z::'a.(P=True --> z=x) & (P=False --> z=y))
Let_def Let s f == f s
Least_def Least P == @x. P(x) & (ALL y. P(y) --> x <= y)

Figure 2.4: The HOL definitions

The definitions above should never be expanded and are shown for completeness only. Instead users should reason in terms of the derived rules shown below or, better still, using high-level tactics.

Some of the rules mention type variables; for example, refl mentions the type variable 'a. This allows you to instantiate type variables explicitly by calling res_inst_tac.

Some derived rules are shown in Figures 2.5 and 2.6, with their ML names. These include natural rules for the logical connectives, as well as sequent-style elimination rules for conjunctions, implications, and universal quantifiers.

Note the equality rules: ssubst performs substitution in backward proofs, while box_equals supports reasoning by simplifying both sides of an equation.

The following simple tactics are occasionally useful:

strip_tac i applies allI and impI repeatedly to remove all outermost universal quantifiers and implications from subgoal i.

case_tac "P" i performs case distinction on P for subgoal i: the latter is replaced by two identical subgoals with the added assumptions P and ~P, respectively.

smp_tac j i applies j times spec and then mp in subgoal i, which is typically useful when forward-chaining from an induction hypothesis. As a generalization of mp_tac, if there are assumptions ∀x. P x → Q x and P a, (x being a vector of j variables) then it replaces the universally quantified implication by Q a. It may instantiate unknowns. It fails if it can do nothing.
CHAPTER 2. HIGHER-ORDER LOGIC

sym  \( s = t \implies t = s \)
trans \( [\{ r = s; s = t \} \implies r = t \)
ssubst \( [\{ t = s; P s \} \implies P t \)
box_equals \( [\{ a = b; a = c; b = d \} \implies c = d \)
arg_cong \( x = y \implies f x = f y \)
fun_cong \( f = g \implies f x = g x \)
cong \( [\{ f = g; x = y \} \implies f x = g y \)
not_sym \( t = s \implies s = t \)

EQUALITY

TrueI True
FalseE False \implies P
conjI \( [\{ P; Q \} \implies P \& Q \)
conjunct1 \( [\{ P \& Q \} \implies P \)
conjunct2 \( [\{ P \& Q \} \implies Q \)
conjE \( [\{ P \& Q; [\{ P; Q \} \implies R \} \implies R \)
disjI1 \( P \implies P \| Q \)
disjI2 \( Q \implies P \| Q \)
disjE \( [\{ P \| Q; P \implies R; Q \implies R \} \implies R \)
notI \( (P \implies \text{False}) \implies \neg P \)
notE \( [\{ \neg P; P \} \implies R \)
impE \( [\{ P \implies Q; P \implies R \} \implies Q \implies R \}

PROPOSITIONAL LOGIC

iffI \( [\{ P \implies Q; Q \implies P \} \implies P \equiv Q \)
iffD1 \( [\{ P \equiv Q; P \} \implies Q \)
iffD2 \( [\{ P \equiv Q; Q \} \implies P \)
iffE \( [\{ P \equiv Q; [\{ P \implies Q; Q \implies P \} \implies R \} \implies R \}

LOGICAL EQUIVALENCE

Figure 2.5: Derived rules for HOL
CHAPTER 2. HIGHER-ORDER LOGIC

allI \( \forall x. P(x) \implies \forall x. P(x) \)

spec \( \exists x. P(x) \implies P(x) \)

allE \[ [\forall x. P(x); P(x) \implies R] \implies R \]

all_dupE \[ [\forall x. P(x); [\forall x. P(x); P(x) \implies R] \implies R \]

exI \( P(x) \implies \exists x. P(x) \)

exE \[ [\exists x. P(x); \forall x. P(x) \implies Q] \implies Q \]

ex1I \[ [\exists! x. P(x); \forall x. P(x) \implies x = a] \implies \exists! x. P(x) \]

ex1E \[ [\exists! x. P(x); \forall x. (P(x) \implies y = x) \implies R] \implies R \]

some_equality \[ [\exists x. P(x); \forall x. P(x) \implies x = a] \implies (\exists x. P(x)) = a \]

QUANTIFIERS AND DESCRIPTIONS

ccontr \( (\neg P \implies False) \implies P \)

classical \( (\neg P \implies P) \implies P \)

excluded_middle \( \forall x. P(x) \lor \neg P(x) \)

disjCI \( (\neg Q \implies P) \implies P \lor Q \)

exCI \( (! x. \neg P(x) \implies P(a)) \implies \exists x. P(x) \)

impCE \[ [\neg P \implies R; \neg P \implies Q] \implies R \]

iffCE \[ [\neg P \implies Q; P \lor Q] \implies R; [Q \implies \neg P \lor \neg Q] \implies R \]

notnotD \( \neg \neg P \implies P \)

swap \( \neg P \implies (\neg Q \implies P) \implies Q \)

CLASSICAL LOGIC

if_P \( P \implies (if P \then x \else y) = x \)

if_not_P \( \neg P \implies (if P \then x \else y) = y \)

split_if \( P(if Q \then x \else y) = ((Q \implies P(x)) \& (\neg Q \implies P(y)) \)

CONDITIONALS

Figure 2.6: More derived rules
CHAPTER 2. HIGHER-ORDER LOGIC

2.3 A formulation of set theory

Historically, higher-order logic gives a foundation for Russell and Whitehead’s theory of classes. Let us use modern terminology and call them sets, but note that these sets are distinct from those of ZF set theory, and behave more like ZF classes.

- Sets are given by predicates over some type $\sigma$. Types serve to define universes for sets, but type-checking is still significant.
- There is a universal set (for each type). Thus, sets have complements, and may be defined by absolute comprehension.
- Although sets may contain other sets as elements, the containing set must have a more complex type.

Finite unions and intersections have the same behaviour in HOL as they do in ZF. In HOL the intersection of the empty set is well-defined, denoting the universal set for the given type.

2.3.1 Syntax of set theory

HOL’s set theory is called Set. The type $\alpha$ set is essentially the same as $\alpha \Rightarrow \text{bool}$. The new type is defined for clarity and to avoid complications involving function types in unification. The isomorphisms between the two types are declared explicitly. They are very natural: Collect maps $\alpha \Rightarrow \text{bool}$ to $\alpha$ set, while op : maps in the other direction (ignoring argument order).

Figure 2.7 lists the constants, infixes, and syntax translations. Figure 2.8 presents the grammar of the new constructs. Infix operators include union and intersection ($A \cup B$ and $A \cap B$), the subset and membership relations, and the image operator ‘’. Note that $a^-:b$ is translated to $\neg(a \in b)$.

The \{a, \ldots\} notation abbreviates finite sets constructed in the obvious manner using insert and {}: \[
\{ a, b, c \} \equiv \text{insert} a(\text{insert} b(\text{insert} c ))
\]

The set \{x. P[x]\} consists of all x (of suitable type) that satisfy $P[x]$, where $P[x]$ is a formula that may contain free occurrences of x. This syntax expands to Collect($\lambda x . P[x]$). It defines sets by absolute comprehension, which is impossible in ZF; the type of x implicitly restricts the comprehension.
name | meta-type | description
--- | --- | ---
{} | $\alpha$ set | the empty set
insert | $[\alpha, \alpha$ set] $\Rightarrow$ $\alpha$ set | insertion of element
Collect | $(\alpha \Rightarrow \text{bool}) \Rightarrow \alpha$ set | comprhension
INTER | $[\alpha$ set, $\alpha \Rightarrow \beta$ set] $\Rightarrow$ $\beta$ set | intersection over a set
UNION | $[\alpha$ set, $\alpha \Rightarrow \beta$ set] $\Rightarrow$ $\beta$ set | union over a set
Inter | $(\alpha$ set)set $\Rightarrow$ $\alpha$ set | set of sets intersection
Union | $(\alpha$ set)set $\Rightarrow$ $\alpha$ set | set of sets union
Pow | $\alpha$ set $\Rightarrow$ $(\alpha$ set)set | powerset
range | $(\alpha \Rightarrow \beta) \Rightarrow \beta$ set | range of a function
Ball Bex | $[\alpha$ set, $\alpha \Rightarrow \text{bool}] \Rightarrow \text{bool}$ | bounded quantifiers

### Constants

<table>
<thead>
<tr>
<th>symbol</th>
<th>name</th>
<th>meta-type</th>
<th>priority</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT</td>
<td>INTER1</td>
<td>$(\alpha \Rightarrow \beta$ set) $\Rightarrow$ $\beta$ set</td>
<td>10</td>
<td>intersection</td>
</tr>
<tr>
<td>UN</td>
<td>UNION1</td>
<td>$(\alpha \Rightarrow \beta$ set) $\Rightarrow$ $\beta$ set</td>
<td>10</td>
<td>union</td>
</tr>
</tbody>
</table>

### Binders

<table>
<thead>
<tr>
<th>symbol</th>
<th>meta-type</th>
<th>priority</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘‘</td>
<td>$[\alpha \Rightarrow \beta, \alpha$ set] $\Rightarrow$ $\beta$ set</td>
<td>Left 90</td>
<td>image</td>
</tr>
</tbody>
</table>
| Int | $[\alpha$ set, $\alpha$ set] $\Rightarrow$ $\alpha$ set | Left 70 | intersection ($\cap$)
| Un | $[\alpha$ set, $\alpha$ set] $\Rightarrow$ $\alpha$ set | Left 65 | union ($\cup$)
| : | $[\alpha, \alpha$ set] $\Rightarrow$ $\text{bool}$ | Left 50 | membership ($\in$)
| <= | $[\alpha$ set, $\alpha$ set] $\Rightarrow$ $\text{bool}$ | Left 50 | subset ($\subseteq$)

### Infixes

Figure 2.7: Syntax of the theory Set
### Translations

<table>
<thead>
<tr>
<th>term</th>
<th>=</th>
<th>other terms...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>{}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>{term, \text{term}}^{*}}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>{id . formula}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>term \text{ Int} term</td>
<td></td>
</tr>
<tr>
<td></td>
<td>term \text{ Un} term</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INT id:term . term</td>
<td></td>
</tr>
<tr>
<td></td>
<td>UN id:term . term</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INT id id^{*}. term</td>
<td></td>
</tr>
<tr>
<td></td>
<td>UN id id^{*}. term</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>formula</th>
<th>=</th>
<th>other formulae...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>term : term</td>
<td></td>
</tr>
<tr>
<td></td>
<td>term \text{ ~:} term</td>
<td></td>
</tr>
<tr>
<td></td>
<td>term \text{ &lt;=} term</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ALL id:term . formula</td>
<td>! id:term . formula</td>
</tr>
<tr>
<td></td>
<td>EX id:term . formula</td>
<td>? id:term . formula</td>
</tr>
</tbody>
</table>

\textbf{Full Grammar}

Figure 2.8: Syntax of the theory \textit{Set} (continued)
The set theory defines two bounded quantifiers:

\[ \forall x \in A. P[x] \] abbreviates \[ \forall x. x \in A \rightarrow P[x] \]

\[ \exists x \in A. P[x] \] abbreviates \[ \exists x. x \in A \land P[x] \]

The constants \( \text{Ball} \) and \( \text{Bex} \) are defined accordingly. Instead of \( \text{Ball} A \ P \) and \( \text{Bex} A \ P \) we may write \( \text{ALL} x:A. P[x] \) and \( \text{EX} x:A. P[x] \). The original notation of Gordon’s HOL system is supported as well: \( ! \) and \( ? \).

Unions and intersections over sets, namely \( \bigcup_{x \in A} B[x] \) and \( \bigcap_{x \in A} B[x] \), are written \( \text{UN} x:A. B[x] \) and \( \text{INT} x:A. B[x] \).

Unions and intersections over types, namely \( \bigcup B[x] \) and \( \bigcap B[x] \), are written \( \text{UN} x. B[x] \) and \( \text{INT} x. B[x] \). They are equivalent to the previous union and intersection operators when \( A \) is the universal set.

The operators \( \bigcup A \) and \( \bigcap A \) act upon sets of sets. They are not binders, but are equal to \( \bigcup_{x \in A} x \) and \( \bigcap_{x \in A} x \), respectively.

### 2.3.2 Axioms and rules of set theory

Figure 2.9 presents the rules of theory Set. The axioms \text{mem_Collect_eq} and \text{Collect_mem_eq} assert that the functions \text{Collect} and \text{op} : are isomorphisms. Of course, \text{op} : also serves as the membership relation.
CHAPTER 2. HIGHER-ORDER LOGIC

CollectI \[ [\forall x. x:A \Rightarrow P x] \Rightarrow a : \{x. P x\} \]
CollectD \[ [a : \{x. P x\}] \Rightarrow P a \]
CollectE \[ [a : \{x. P x\}; P a \Rightarrow W] \Rightarrow W \]

ballI \[ [\forall x. x:A \Rightarrow P x] \Rightarrow \exists x:A. P x \]
bspec \[ [\forall x:A. P x; x:A] \Rightarrow P x \]
ballE \[ [\forall x:A. P x; P x \Rightarrow Q; \neg x:A \Rightarrow Q] \Rightarrow Q \]

bexI \[ [P x; x:A] \Rightarrow \exists x:A. P x \]
bexCI \[ [\forall x:A. \neg P x \Rightarrow P a; a:A] \Rightarrow \exists x:A. P x \]
bexE \[ [\exists x:A. P x; \forall x. [P x \Rightarrow Q] \Rightarrow Q] \Rightarrow Q \]

Comprehension and Bounded quantifiers

subsetI \[ (\forall x. x:A \Rightarrow x:B) \Rightarrow A \subseteq B \]
subsetD \[ [A \subseteq B; c:A] \Rightarrow c:B \]
subsetCE \[ [A \subseteq B; \neg (c:A) \Rightarrow P; c:B \Rightarrow P] \Rightarrow P \]

subset_refl \[ A \subseteq A \]
subset_trans \[ [A \subseteq B; B \subseteq C] \Rightarrow A \subseteq C \]

equalityI \[ [A \subseteq B; B \subseteq A] \Rightarrow A = B \]
equalityD1 \[ A = B \Rightarrow A \subseteq B \]
equalityD2 \[ A = B \Rightarrow B \subseteq A \]
equalityE \[ [A = B; [A \subseteq B; B \subseteq A] \Rightarrow P] \Rightarrow P \]
equalityCE \[ [A = B; [c:A; c:B] \Rightarrow P; [\neg c:A; \neg c:B] \Rightarrow P] \Rightarrow P \]

The subset and equality relations

Figure 2.10: Derived rules for set theory
emptyE \ a \ : \ {} \ ==\to \ P

insertI1 \ a \ : \ insert \ a \ B
insertI2 \ a \ : \ B \ ==\to \ a \ : \ insert \ b \ B
insertE \ [| \ a \ : \ insert \ b \ A; \ a=b \ ==\to \ P; \ a:A \ ==\to \ P |] \ ==\to \ P

ComplI \ [| \ c:A \ ==\to \ False |] \ ==\to \ c \ : \ -A
ComplD \ [| \ c \ : \ -A |] \ ==\to \ "c:A

UnI1 \ c:A \ ==\to \ c \ : \ A \ Un \ B
UnI2 \ c:B \ ==\to \ c \ : \ A \ Un \ B
UnCI \ ("c:B \ ==\to \ c:A) \ ==\to \ c \ : \ A \ Un \ B
UnE \ [| \ c \ : \ A \ Un \ B; \ c:A \ ==\to \ P; \ c:B \ ==\to \ P |] \ ==\to \ P

IntI \ [| \ c:A; \ c:B |] \ ==\to \ c \ : \ A \ Int \ B
IntD1 \ c \ : \ A \ Int \ B \ ==\to \ c:A
IntD2 \ c \ : \ A \ Int \ B \ ==\to \ c:B
IntE \ [| \ c \ : \ A \ Int \ B; \ [| \ c:A; \ c:B |] \ ==\to \ P |] \ ==\to \ P

UN_I \ [| \ a:A; \ b:B \ a |] \ ==\to \ b: (UN x:A. B x)
UN_E \ [| \ b: (UN x:A. B x); \ !!x.[| \ x:A; \ b:B \ x |] \ ==\to \ R |] \ ==\to \ R

INT_I \ (!!x. x:A \ ==\to \ b: B x) \ ==\to \ b: (INT x:A. B x)
INT_D \ [| \ b: (INT x:A. B x); \ a:A |] \ ==\to \ b: B a
INT_E \ [| \ b: (INT x:A. B x); \ b: B a \ ==\to \ R; \ "a:A \ ==\to \ R |] \ ==\to \ R

UnionI \ [| \ X:C; \ A:X |] \ ==\to \ A : Union C
UnionE \ [| \ A : Union C; \ !!X.[| \ A:X; \ X:C |] \ ==\to \ R |] \ ==\to \ R

InterI \ [| \ !!X. X:C \ ==\to \ A:X |] \ ==\to \ A : Inter C
InterD \ [| \ A : Inter C; \ X:C |] \ ==\to \ A:X
InterE \ [| \ A : Inter C; \ A:X \ ==\to \ R; \ "X:C \ ==\to \ R |] \ ==\to \ R

PowI \ A<=B \ ==\to \ A: Pow B
PowD \ A: Pow B \ ==\to \ A<=B

imageI \ [| \ x:A |] \ ==\to \ f \ x : f"A
imageE \ [| \ b : f"A; \ !!x.[| \ b=f \ x; \ x:A |] \ ==\to \ P |] \ ==\to \ P

rangeI \ f \ x : range f
rangeE \ [| \ b : range f; \ !!x.[| \ b=f \ x |] \ ==\to \ P |] \ ==\to \ P

Figure 2.11: Further derived rules for set theory
CHAPTER 2. HIGHER-ORDER LOGIC

All the other axioms are definitions. They include the empty set, bounded quantifiers, unions, intersections, complements and the subset relation. They also include straightforward constructions on functions: image (""") and range.

Figures 2.10 and 2.11 present derived rules. Most are obvious and resemble rules of Isabelle’s ZF set theory. Certain rules, such as subsetCE, bexCI and UnCI, are designed for classical reasoning; the rules subsetD, bexI, Un1 and Un2 are not strictly necessary but yield more natural proofs. Similarly, equalityCE supports classical reasoning about extensionality, after the fashion of iffCE. See the file HOL/Set.ML for proofs pertaining to set theory.

Figure 2.12 presents lattice properties of the subset relation. Unions form least upper bounds; non-empty intersections form greatest lower bounds. Reasoning directly about subsets often yields clearer proofs than reasoning about the membership relation. See the file HOL/subset.ML.

Figure 2.13 presents many common set equalities. They include commutative, associative and distributive laws involving unions, intersections and complements. For a complete listing see the file HOL/equalities.ML.

! Blast_tac proves many set-theoretic theorems automatically. Hence you seldom need to refer to the theorems above.

2.3.3 Properties of functions

Figure 2.14 presents a theory of simple properties of functions. Note that inv f uses Hilbert’s ε to yield an inverse of f. See the file HOL/Fun.ML for a complete listing of the derived rules. Reasoning about function composition
CHAPTER 2. HIGHER-ORDER LOGIC

Int_absorb \ A \ Int \ A = A
Int_commute \ A \ Int \ B = B \ Int \ A
Int_assoc \ (A \ Int \ B) \ Int \ C = A \ Int \ (B \ Int \ C)
Int_Un_distrib \ (A \ Un \ B) \ Int \ C = (A \ Int \ C) \ Un \ (B \ Int \ C)

Un_absorb \ A \ Un \ A = A
Un_commute \ A \ Un \ B = B \ Un \ A
Un_assoc \ (A \ Un \ B) \ Un \ C = A \ Un \ (B \ Un \ C)
Un_Int_distrib \ (A \ Int \ B) \ Un \ C = (A \ Un \ C) \ Int \ (B \ Un \ C)

Compl_disjoint \ A \ Int \ (-A) = \{x. False\}
Compl_partition \ A \ Un \ (-A) = \{x. True\}
double_complement \ -(-A) = A
Compl_Un \ -(A \ Un \ B) = (-A) \ Int \ (-B)
Compl_Int \ -(A \ Int \ B) = (-A) \ Un \ (-B)

Union_Un_distrib \ Union(A \ Un \ B) = (Union A) \ Un \ (Union B)
Int_Union \ A \ Int \ (Union B) = (UN \ C:B. A \ Int \ C)

Inter_Un_distrib \ Inter(A \ Un \ B) = (Inter A) \ Int \ (Inter B)
Un_Inter \ A \ Un \ (Inter B) = (INT \ C:B. A \ Un \ C)

Figure 2.13: Set equalities

<table>
<thead>
<tr>
<th>name</th>
<th>meta-type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inj</td>
<td>(\alpha \Rightarrow \beta) \Rightarrow bool</td>
<td>injective/surjective</td>
</tr>
<tr>
<td>surj</td>
<td>[\alpha \Rightarrow \beta, \alpha \text{ set}] \Rightarrow bool</td>
<td>injective over subset</td>
</tr>
<tr>
<td>inj_on</td>
<td>(\alpha \Rightarrow \beta \Rightarrow \beta \Rightarrow \alpha)</td>
<td>inverse function</td>
</tr>
</tbody>
</table>

inj_def \ inj f \ == \ ! x y. f x=f y \ --> x=y
surj_def \ surj f \ == \ ! y. ? x. y=f x
inj_on_def \ inj_on f A \ == \ !x:A. !y:A. f x=f y \ --> x=y
inv_def \ inv f \ == \ (%y. @x. f(x)=y)

Figure 2.14: Theory Fun
(the operator $\circ$) and the predicate $\text{surj}$ is done simply by expanding the definitions.

There is also a large collection of monotonicity theorems for constructions on sets in the file \texttt{HOL/mono.ML}.

### 2.4 Simplification and substitution

Simplification tactics such as \texttt{Asm_simp_tac} and \texttt{Full_simp_tac} use the default simpset (\texttt{simpset()}), which works for most purposes. A quite minimal simplification set for higher-order logic is \texttt{HOL_ss}; even more frugal is \texttt{HOL_basic_ss}. Equality (=), which also expresses logical equivalence, may be used for rewriting. See the file \texttt{HOL/simpdata.ML} for a complete listing of the basic simplification rules.

See the \textit{Reference Manual} for details of substitution and simplification.

Reducing $a = b \land P(a)$ to $a = b \land P(b)$ is sometimes advantageous. The left part of a conjunction helps in simplifying the right part. This effect is not available by default: it can be slow. It can be obtained by including \texttt{conj_cong} in a simpset, \texttt{addcongs [conj_cong]}.

By default only the condition of an \texttt{if} is simplified but not the \texttt{then} and \texttt{else} parts. Of course the latter are simplified once the condition simplifies to \texttt{True} or \texttt{False}. To ensure full simplification of all parts of a conditional you must remove \texttt{if_weak_cong} from the simpset, \texttt{delcongs [if_weak_cong]}.

If the simplifier cannot use a certain rewrite rule — either because of nontermination or because its left-hand side is too flexible — then you might try \texttt{stac}:

\begin{verbatim}
stac thm i, where thm is of the form lhs = rhs, replaces in subgoal i instances of lhs by corresponding instances of rhs. In case of multiple instances of lhs in subgoal i, backtracking may be necessary to select the desired ones.

If thm is a conditional equality, the instantiated condition becomes an additional (first) subgoal.
\end{verbatim}

\texttt{HOL} provides the tactic \texttt{hyp_subst_tac}, which substitutes for an equality throughout a subgoal and its hypotheses. This tactic uses \texttt{HOL}'s general substitution rule.
2.4.1 Case splitting

HOL also provides convenient means for case splitting during rewriting. Goals containing a subterm of the form if b then...else... often require a case distinction on b. This is expressed by the theorem split_if:

\[ \forall P (\text{if } b \text{ then } x \text{ else } y) = ((\forall b \rightarrow P(x)) \land (\neg b \rightarrow P(y))) \]  

For example, a simple instance of (*) is

\[ x \in (\text{if } x \in A \text{ then } A \text{ else } \{x\}) = ((x \in A \rightarrow x \in A) \land (x \notin A \rightarrow x \in \{x\}) \]

Because (*) is too general as a rewrite rule for the simplifier (the left-hand side is not a higher-order pattern in the sense of the Reference Manual), there is a special infix function addsplits of type simpset * thm list -> simpset (analogous to addsimps) that adds rules such as (*) to a simpset, as in

by(simp_tac (simpset() addsplits [split_if]) 1);

The effect is that after each round of simplification, one occurrence of if is split according to split_if, until all occurrences of if have been eliminated.

It turns out that using split_if is almost always the right thing to do. Hence split_if is already included in the default simpset. If you want to delete it from a simpset, use delsplits, which is the inverse of addsplits:

by(simp_tac (simpset() delsplits [split_if]) 1);

In general, addsplits accepts rules of the form

\[ \forall P (c x_1 \ldots x_n) = rhs \]

where c is a constant and rhs is arbitrary. Note that (*) is of the right form because internally the left-hand side is \( \forall P(\text{if } b x y) \). Important further examples are splitting rules for case expressions (see §2.5.4 and §2.6.1).

Analogous to Addsimps and Delsimsps, there are also imperative versions of addsplits and delsplits

Add splits: thm list -> unit
Delsplits: thm list -> unit

for adding splitting rules to, and deleting them from the current simpset.

2.5 Types

This section describes HOL’s basic predefined types (\( \alpha \times \beta \), \( \alpha + \beta \), nat and \( \alpha \ list \)) and ways for introducing new types in general. The most important type construction, the datatype, is treated separately in §2.6.
2.5.1 Product and sum types

<table>
<thead>
<tr>
<th>symbol</th>
<th>meta-type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pair</td>
<td>$[\alpha, \beta] \Rightarrow \alpha \times \beta$</td>
<td>ordered pairs $(a, b)$</td>
</tr>
<tr>
<td>fst</td>
<td>$\alpha \times \beta \Rightarrow \alpha$</td>
<td>first projection</td>
</tr>
<tr>
<td>snd</td>
<td>$\alpha \times \beta \Rightarrow \beta$</td>
<td>second projection</td>
</tr>
<tr>
<td>split</td>
<td>$[[\alpha, \beta] \Rightarrow \gamma, \alpha \times \beta] \Rightarrow \gamma$</td>
<td>generalized projection</td>
</tr>
<tr>
<td>Sigma</td>
<td>$[\alpha \text{ set}, \alpha \Rightarrow \beta \text{ set}] \Rightarrow (\alpha \times \beta)\text{ set}$</td>
<td>general sum of sets</td>
</tr>
</tbody>
</table>

Sigma_def  \quad Sigma \ A \ B \ = \ \text{UN x:A. UN y:B x. \{(x,y)\}}

prod.inject \quad ((a,b) = (a',b')) = (a=a' & b=b')
Pair_inject \quad \forall \ (a, b) = (a',b'); \forall \ (a=a'; b=b') \Rightarrow R \ |\ R \Rightarrow R
prod.exhaust \quad \forall \ x y. p = (x,y) \Rightarrow Q \ |\ Q \Rightarrow Q

fst_conv    \quad \text{fst (a,b) = a}
snd_conv    \quad \text{snd (a,b) = b}
surjective_pairing \quad p = (\text{fst p, snd p})

split      \quad \text{case_prod c (a,b) = c a b}
prod.split  \quad R(\text{case_prod c p}) = (! x y. p = (x,y) \Rightarrow R(c x y))

SigmaI \quad \forall \ a:A; \ b:B \ a \ |\ \Rightarrow (a,b) : \text{Sigma A B}
SigmaE \quad \forall \ c: \text{Sigma A B}; \forall \ x y. [\forall \ x:A; \ y:B \ x; \ c=(x,y) \ |\ ] \Rightarrow P \ |\ [] \Rightarrow P

Figure 2.15: Type $\alpha \times \beta$

Theory \texttt{Prod} (Fig. 2.15) defines the product type $\alpha \times \beta$, with the ordered pair syntax $(a, b)$. General tuples are simulated by pairs nested to the right:

\[
\begin{array}{c|c}
\text{external} & \text{internal} \\
\tau_1 \times \ldots \times \tau_n & \tau_1 \times (\ldots (\tau_{n-1} \times \tau_n)\ldots) \\
(t_1, \ldots, t_n) & (t_1, (\ldots, (t_{n-1}, t_n)\ldots)
\end{array}
\]

In addition, it is possible to use tuples as patterns in abstractions:

\%($x,y$). $t$ stands for \texttt{split}(%$x\ y$. $t$)

Nested patterns are also supported. They are translated stepwise:

\%($x,y,z$). $t$ $\leadsto$ \%($x,(y,z)$). $t$
$\leadsto$ \texttt{case_prod}(%$x$,\%($y,z$). $t$)
$\leadsto$ \texttt{case_prod}(%$x$. \texttt{case_prod}(%$y\ z$. $t$))

The reverse translation is performed upon printing.
The translation between patterns and `split` is performed automatically by the parser and printer. Thus the internal and external form of a term may differ, which can affects proofs. For example the term \((\% (x,y). (y,x))(a,b)\) requires the theorem `split` (which is in the default simpset) to rewrite to \((b,a)\).

In addition to explicit \(\lambda\)-abstractions, patterns can be used in any variable binding construct which is internally described by a \(\lambda\)-abstraction. Some important examples are

**Let:** `let pattern = t in u`

**Quantifiers:** `ALL pattern:A. P`

**Choice:** `SOME pattern. P`

**Set operations:** `UN pattern:A. B`

**Sets:** \{pattern. P\}

There is a simple tactic which supports reasoning about patterns:

`split_all_tac` \(i\) replaces in subgoal \(i\) all `!!`-quantified variables of product type by individual variables for each component. A simple example:

1. `!!p. (%(x,y,z). (x, y, z)) p = p`
   by(`split_all_tac 1`);
1. `!!x xa ya. (%(x,y,z). (x, y, z)) (x, xa, ya) = (x, xa, ya)`

Theory `Prod` also introduces the degenerate product type `unit` which contains only a single element named `()` with the property

\[
\text{unit_eq } u = ()
\]

Theory `Sum` (Fig. 2.16) defines the sum type \(\alpha + \beta\) which associates to the right and has a lower priority than `*`: \(\tau_1 + \tau_2 + \tau_3 * \tau_4\) means \(\tau_1 + (\tau_2 + (\tau_3 * \tau_4))\).

The definition of products and sums in terms of existing types is not shown. The constructions are fairly standard and can be found in the respective theory files. Although the sum and product types are constructed manually for foundational reasons, they are represented as actual datatypes later.
### Symbol Meta-type Description

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meta-type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{Inl}</td>
<td>( \alpha \Rightarrow \alpha + \beta )</td>
<td>first injection</td>
</tr>
<tr>
<td>\text{Inr}</td>
<td>( \beta \Rightarrow \alpha + \beta )</td>
<td>second injection</td>
</tr>
<tr>
<td>\text{case_sum}</td>
<td>( [\alpha \Rightarrow \gamma, \beta \Rightarrow \gamma, \alpha + \beta \Rightarrow \gamma] )</td>
<td>conditional</td>
</tr>
</tbody>
</table>

\begin{align*}
\text{Inl\_not\_Inr} & \quad \text{Inl } a \neq \text{Inr } b \\
\text{inj\_Inl} & \quad \text{inj } \text{Inl} \\
\text{inj\_Inr} & \quad \text{inj } \text{Inr} \\
\text{sumE} & \quad [\! \forall x. \ P(\text{Inl } x); \ \forall y. \ P(\text{Inr } y) ] \Rightarrow P \ s \\
\text{case\_sum\_Inl} & \quad \text{case\_sum } f \ g (\text{Inl } x) = f \ x \\
\text{case\_sum\_Inr} & \quad \text{case\_sum } f \ g (\text{Inr } x) = g \ x \\
\text{surjective\_sum} \ & \quad \text{case\_sum } (\%x. \ f(\text{Inl } x)) (\%y. \ f(\text{Inr } y)) \ s = f \ s \\
\text{sum\_split\_case} \ & \quad \text{R(\text{case\_sum } f \ g \ s) = ((\! x. \ s = \text{Inl}(x) \Rightarrow R(f(x))) \ & \ (\! y. \ s = \text{Inr}(y) \Rightarrow R(g(y))))}
\end{align*}

Figure 2.16: Type \( \alpha + \beta \)

#### 2.5.2 The type of natural numbers, \textit{nat}

The theory Nat defines the natural numbers in a roundabout but traditional way. The axiom of infinity postulates a type \textit{ind} of individuals, which is non-empty and closed under an injective operation. The natural numbers are inductively generated by choosing an arbitrary individual for 0 and using the injective operation to take successors. This is a least fixedpoint construction.

Type \textit{nat} is an instance of class \textit{ord}, which makes the overloaded functions of this class (especially \(<\) and \(\leq\), but also \textit{min}, \textit{max} and \textit{LEAST}) available on \textit{nat}. Theory Nat also shows that \(\leq\) is a linear order, so \textit{nat} is also an instance of class \textit{linorder}.

Theory NatArith develops arithmetic on the natural numbers. It defines addition, multiplication and subtraction. Theory Divides defines division, remainder and the “divides” relation. The numerous theorems proved include commutative, associative, distributive, identity and cancellation laws. See Figs. 2.17 and 2.18. The recursion equations for the operators \(\oplus\), \(-\) and \(*\) on \textit{nat} are part of the default simpset.

Functions on \textit{nat} can be defined by primitive or well-founded recursion; see §2.7. A simple example is addition. Here, \texttt{op +} is the name of the infix operator \(+\), following the standard convention.
CHAPTER 2. HIGHER-ORDER LOGIC

symbol  meta-type  priority  description
0        α         0          zero
Suc      nat ⇒ nat successor function
*        [α,α] ⇒ α  Left 70  multiplication
div      [α,α] ⇒ α  Left 70  division
mod      [α,α] ⇒ α  Left 70  modulus
dvd      [α,α] ⇒ bool Left 70 “divides” relation
+        [α,α] ⇒ α  Left 65  addition
-        [α,α] ⇒ α  Left 65  subtraction

Constants and infixes

nat_induct  [| P 0; !!n. P n ==> P(Suc n) |] ==> P n
Suc_not_Zero Suc m~0
inj_Suc      inj Suc
n_not_Suc_n  n~Suc n

Basic properties

Figure 2.17: The type of natural numbers, nat

0+n = n
(Suc m)+n = Suc(m+n)

m-0 = m
0-n = n
Suc(m)-Suc(n) = m-n

0*n = 0
Suc(m)*n = n + m*n

mod_less  m<n ==> m mod n = m
mod_geq   [| 0<n; ~m<n |] ==> m mod n = (m-n) mod n

div_less  m<n ==> m div n = 0
div_geq   [| 0<n; ~m<n |] ==> m div n = Suc((m-n) div n)

Figure 2.18: Recursion equations for the arithmetic operators
CHAPTER 2. HIGHER-ORDER LOGIC

primrec
"0 + n = n"
"Suc m + n = Suc (m + n)"

There is also a case-construct of the form

case e of 0 => a | Suc m => b

Note that Isabelle insists on precisely this format; you may not even change the order of the two cases. Both primrec and case are realized by a recursion operator rec_nat, which is available because nat is represented as a datatype.

Tactic induct_tac "n" i performs induction on variable n in subgoal i using theorem nat_induct. There is also the derived theorem less_induct:

[| !!n. [| ! m. m<n --> P m |] ==> P n |] ==> P n

2.5.3 Numerical types and numerical reasoning

The integers (type int) are also available in HOL, and the reals (type real) are available in the logic image HOL-Complex. They support the expected operations of addition (+), subtraction (-) and multiplication (*), and much else. Type int provides the div and mod operators, while type real provides real division and other operations. Both types belong to class linorder, so they inherit the relational operators and all the usual properties of linear orderings. For full details, please survey the theories in subdirectories Integ, Real, and Complex.

All three numeric types admit numerals of the form sd...d, where s is an optional minus sign and d...d is a string of digits. Numerals are represented internally by a datatype for binary notation, which allows numerical calculations to be performed by rewriting. For example, the integer division of 54342339 by 3452 takes about five seconds. By default, the simplifier cancels like terms on the opposite sites of relational operators (reducing z+x<x+y to z<y, for instance. The simplifier also collects like terms, replacing x+y+x*3 by 4*x+y.

Sometimes numerals are not wanted, because for example n+3 does not match a pattern of the form Suc k. You can re-arrange the form of an arithmetic expression by proving (via subgoal_tac) a lemma such as n+3 = Suc (Suc (Suc n)). As an alternative, you can disable the fancier simplifications by using a basic simpset such as HOL_ss rather than the default one, simpset().

Reasoning about arithmetic inequalities can be tedious. Fortunately, HOL provides a decision procedure for linear arithmetic: formulae involving addition and subtraction. The simplifier invokes a weak version of this
decision procedure automatically. If this is not sufficient, you can invoke the full procedure \texttt{Lin_Arith.tac} explicitly. It copes with arbitrary formulae involving $=, <, \leq, +, -, \text{Suc, min, max}$ and numerical constants. Other subterms are treated as atomic, while subformulae not involving numerical types are ignored. Quantified subformulae are ignored unless they are positive universal or negative existential. The running time is exponential in the number of occurrences of \texttt{min, max}, and $-$ because they require case distinctions. If $k$ is a numeral, then \texttt{div} $k$, \texttt{mod} $k$ and \texttt{k dvd} are also supported. The former two are eliminated by case distinctions, again blowing up the running time. If the formula involves explicit quantifiers, \texttt{Lin_Arith.tac} may take super-exponential time and space.

If \texttt{Lin_Arith.tac} fails, try to find relevant arithmetic results in the library. The theories \texttt{Nat} and \texttt{NatArith} contain theorems about $<, \leq, +, -$ and $\ast$. Theory \texttt{Divides} contains theorems about \texttt{div} and \texttt{mod}. Use Proof General’s \texttt{find} button (or other search facilities) to locate them.

### 2.5.4 The type constructor for lists, \texttt{list}

Figure 2.19 presents the theory \texttt{List}: the basic list operations with their types and syntax. Type $\alpha \text{ list}$ is defined as a \texttt{datatype} with the constructors $\texttt{[]}$ and $\texttt{#}$. As a result the generic structural induction and case analysis tactics \texttt{induct_tac} and \texttt{cases_tac} also become available for lists. A case construct of the form

\[
\text{case } e \text{ of } \texttt{[]} => a | x \# xs => b
\]

is defined by translation. For details see §2.6. There is also a case splitting rule \texttt{list.split}

\[
P(\text{case } e \text{ of } \texttt{[]} => a | x \# xs => f x xs) = \\
((e = \texttt{[]} \to P(a)) \land (\forall x \hspace{1pt} xs. \hspace{1pt} e = x \# xs \to P(f x xs)))
\]

which can be fed to \texttt{addssplits} just like \texttt{split_if} (see §2.4.1).

\texttt{List} provides a basic library of list processing functions defined by primitive recursion. The recursion equations are shown in Figs. 2.20 and 2.21.

### 2.6 Datatype definitions

Inductive datatypes, similar to those of \texttt{ML}, frequently appear in applications of Isabelle/HOL. In principle, such types could be defined by hand via \texttt{typedef}, but this would be far too tedious. The \texttt{datatype} definition
### Constants and infixes

<table>
<thead>
<tr>
<th>External</th>
<th>Internal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[x_1, \ldots, x_n]</td>
<td>\texttt{x_1 # \cdots # x_n #[]}</td>
<td>finite list</td>
</tr>
<tr>
<td>[x:l. \ P]</td>
<td>\texttt{filter (\lambda x.P) l}</td>
<td>list comprehension</td>
</tr>
</tbody>
</table>

### Translations

Figure 2.19: The theory List
null [] = True
null (x#xs) = False

hd (x#xs) = x
tl (x#xs) = xs
tl [] = []

[] @ ys = ys
(x#xs) @ ys = x # xs @ ys

set [] = {}
set (x#xs) = insert x (set xs)

x mem [] = False
x mem (y#ys) = (if y=x then True else x mem ys)

concat([]) = []
concat(x#xs) = x @ concat(xs)

rev([]) = []
rev(x#xs) = rev(xs) @ [x]

length([]) = 0
length(x#xs) = Suc(length(xs))

xs!0 = hd xs
xs!(Suc n) = (tl xs)!n

Figure 2.20: Simple list processing functions
map f [] = []
map f (x#xs) = f x # map f xs

filter P [] = []
filter P (x#xs) = (if P x then x#filter P xs else filter P xs)

foldl f a [] = a
foldl f a (x#xs) = foldl f (f a x) xs

take n [] = []
take n (x#xs) = (case n of 0 => [] | Suc(m) => x # take m xs)

drop n [] = []
drop n (x#xs) = (case n of 0 => x#xs | Suc(m) => drop m xs)

takeWhile P [] = []
takeWhile P (x#xs) = (if P x then x#takeWhile P xs else [])

dropWhile P [] = []
dropWhile P (x#xs) = (if P x then dropWhile P xs else xs)

Figure 2.21: Further list processing functions

The current datatype package can handle both mutual and indirect recursion. It also offers to represent existing types as datatypes giving the advantage of a more uniform view on standard theories.

2.6.1 Basics

A general datatype definition is of the following form:

\[
\text{datatype } (\vec{\alpha}) t_1 = C_1^{1} \tau_{1,1}^{1} \cdots \tau_{1,m_1}^{1} \mid \cdots \mid C_{k_1}^{1} \tau_{k_1,1}^{1} \cdots \tau_{k_1,m_{k_1}}^{1} \\
\vdots \\
\text{and } (\vec{\alpha}) t_n = C_1^{n} \tau_{1,1}^{n} \cdots \tau_{1,m_n}^{n} \mid \cdots \mid C_{k_n}^{n} \tau_{k_n,1}^{n} \cdots \tau_{k_n,m_{k_n}}^{n}
\]

where \( \vec{\alpha} = (\alpha_1, \ldots, \alpha_h) \) is a list of type variables, \( C_i^j \) are distinct constructor names and \( \tau_{i,j}^{i'} \) are admissible types containing at most the type variables \( \alpha_1, \ldots, \alpha_h \). A type \( \tau \) occurring in a datatype definition is admissible if and only if
• $\tau$ is non-recursive, i.e. $\tau$ does not contain any of the newly defined type constructors $t_1, \ldots, t_n$, or
• $\tau = (\vec{\alpha})t_j'$ where $1 \leq j' \leq n$, or
• $\tau = (\tau_1', \ldots, \tau_{k'}')t'$, where $t'$ is the type constructor of an already existing datatype and $\tau_1', \ldots, \tau_{k'}'$ are admissible types.
• $\tau = \sigma \rightarrow \tau'$, where $\tau'$ is an admissible type and $\sigma$ is non-recursive (i.e. the occurrences of the newly defined types are strictly positive).

If some $(\vec{\alpha})t_j'$ occurs in a type $\tau_{i,i'}$ of the form

$$(\ldots, (\vec{\alpha})t_j', \ldots, \ldots)t'$$

this is called a nested (or indirect) occurrence. A very simple example of a datatype is the type `list`, which can be defined by

```plaintext
datatype 'a list = Nil
| Cons 'a ('a list)
```

Arithmetic expressions `aexp` and boolean expressions `bexp` can be modelled by the mutually recursive datatype definition

```plaintext
datatype 'a aexp = If.then_else ('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)
| Or ('a bexp) ('a bexp)
```

The datatype `term`, which is defined by

```plaintext
datatype ('a, 'b) term = Var 'a
| App 'b ((('a, 'b) term) list)
```

is an example for a datatype with nested recursion. Using nested recursion involving function spaces, we may also define infinitely branching datatypes, e.g.

```plaintext
datatype 'a tree = Atom 'a | Branch "nat => 'a tree"
```

Types in HOL must be non-empty. Each of the new datatypes $(\vec{\alpha})t_j$ with $1 \leq j \leq n$ is non-empty if and only if it has a constructor $C_j^\vec{\alpha}$ with the following property: for all argument types $\tau_{i,i'}$ of the form $(\vec{\alpha})t_j'$, the datatype $(\vec{\alpha})t_j'$ is non-empty.
If there are no nested occurrences of the newly defined datatypes, obviously at least one of the newly defined datatypes \((\vec{\alpha})t_j\) must have a constructor \(C_j\) without recursive arguments, a base case, to ensure that the new types are non-empty. If there are nested occurrences, a datatype can even be non-empty without having a base case itself. Since \texttt{list} is a non-empty datatype, \texttt{datatype t = C (t list)} is non-empty as well.

**Freeness of the constructors**

The datatype constructors are automatically defined as functions of their respective type:

\[ C_j : [\tau_{j,1}, \ldots, \tau_{j,m_j}] \Rightarrow (\alpha_1, \ldots, \alpha_h)t_j \]

These functions have certain freeness properties. They construct distinct values:

\[ C_j x_1 \ldots x_{m_j} \neq C_{j'} y_1 \ldots y_{m_{j'}} \quad \text{for all } i \neq i'. \]

The constructor functions are injective:

\[ (C_j x_1 \ldots x_{m_j} = C_{j'} y_1 \ldots y_{m_{j'}}) = (x_1 = y_1 \land \ldots \land x_{m_j} = y_{m_{j'}}) \]

Since the number of distinctness inequalities is quadratic in the number of constructors, the datatype package avoids proving them separately if there are too many constructors. Instead, specific inequalities are proved by a suitable simplification procedure on demand.\(^3\)

\(^3\)This procedure, which is already part of the default simpset, may be referred to by the ML identifier \texttt{DatatypePackage.distinct_simproc}.  

Structural induction

The datatype package also provides structural induction rules. For datatypes without nested recursion, this is of the following form:

\[ \forall x_1 \ldots x_{m_1} \cdot \left[ P_{s_1} x_1^1 \ldots x_{i_1}^1 \right] \Rightarrow P_1 \left( C_1^1 x_1 \ldots x_{m_1}^1 \right) \]

\[ \vdots \]

\[ \forall x_1 \ldots x_{m_k^1} \cdot \left[ P_{s_1} x_1^1 \ldots x_{i_1}^1 \right] \Rightarrow P_1 \left( C_{k_1}^1 x_1 \ldots x_{m_k}^1 \right) \]

\[ \vdots \]

\[ \forall x_1 \ldots x_{m_k^n} \cdot \left[ P_{s_1} x_1^1 \ldots x_{i_1}^1 \right] \Rightarrow P_n \left( C_{k_n}^1 x_1 \ldots x_{m_n}^1 \right) \]

\[ \forall x_1 \ldots x_{m_n} \cdot \left[ P_{s_1} x_1^1 \ldots x_{i_1}^1 \right] \Rightarrow \bigwedge P_1 x_1 \land \ldots \land P_n x_n \]

where

\[ Rec_i^j := \left\{ \left( t_{i,1}^j, s_{i,1}^j \right), \ldots, \left( t_{i,\ell_i}^j, s_{i,\ell_i}^j \right) \right\} = \left\{ \left( i', i'' \right) \mid 1 \leq i' \leq m_i^j \land 1 \leq i'' \leq n \land \tau_{i,i'}^j = (\alpha_1, \ldots, \alpha_k) t_{i''} \right\} \]

i.e. the properties \( P_j \) can be assumed for all recursive arguments.

For datatypes with nested recursion, such as the term example from above, things are a bit more complicated. Conceptually, Isabelle/HOL unfolds a definition like

\texttt{datatype ('a,'b) term = Var 'a} \\
\texttt{| App 'b ('('a, 'b) term) list)}

to an equivalent definition without nesting:

\texttt{datatype ('a,'b) term = Var} \\
\texttt{| App 'b ('('a, 'b) term) list)}

and \texttt{('a,'b) term_list = Nil'} \\
\texttt{| Cons' ('('a,'b) term) ('('a,'b) term_list)}

Note however, that the type \texttt{('a,'b) term_list} and the constructors \texttt{Nil'} and \texttt{Cons'} are not really introduced. One can directly work with the original (isomorphic) type \texttt{('a, 'b) term) list} and its existing constructors \texttt{Nil} and \texttt{Cons}. Thus, the structural induction rule for \texttt{term} gets the form

\[ \forall x. P_1 \left( \text{Var} \; x \right) \]

\[ \forall x_1 \; x_2. P_2 \; x_2 \Rightarrow P_1 \left( \text{App} \; x_1 \; x_2 \right) \\
\]

\[ P_2 \; \text{Nil} \]

\[ \forall x_1 \; x_2. \left[ P_1 \; x_1; P_2 \; x_2 \right] \Rightarrow P_2 \left( \text{Cons} \; x_1 \; x_2 \right) \]

\[ P_1 \; x_1 \land P_2 \; x_2 \]
Note that there are two predicates $P_1$ and $P_2$, one for the type $(\tau,'b) \text{ term}$ and one for the type $((\tau, 'b) \text{ term}) \text{ list}$.

For a datatype with function types such as $\tau \text{ tree}$, the induction rule is of the form

$$\forall a. \ P(\text{Atom } a) \land ts . (\forall x . \ P(ts x)) \Longrightarrow P(\text{Branch } ts)$$

In principle, inductive types are already fully determined by freeness and structural induction. For convenience in applications, the following derived constructions are automatically provided for any datatype.

The case construct

The type comes with an ML-like case-construct:

$$\text{case } e \text{ of } C_1^{ij} x_{1,1} \ldots x_{1,m_{i1}^j} \Rightarrow e_1 \ldots$$
$$\ldots | C_{kj}^{ij} x_{k_j,1} \ldots x_{k_j,m_{k_j}^j} \Rightarrow e_{kj}$$

where the $x_{i,j}$ are either identifiers or nested tuple patterns as in §2.5.1.

All constructors must be present, their order is fixed, and nested patterns are not supported (with the exception of tuples). Violating this restriction results in strange error messages.

To perform case distinction on a goal containing a case-construct, the theorem $t_j \text{.split}$ is provided:

$$P(t_j \text{.case } f_1 \ldots f_{kj} e) = ((\forall x_1 \ldots x_{m_i^j} . e = C_1^{ij} x_1 \ldots x_{m_i^j} \rightarrow P(f_1 x_1 \ldots x_{m_i^j}))$$
$$\land \ldots \land$$
$$((\forall x_1 \ldots x_{m_{kj}^j} . e = C_{kj}^{ij} x_1 \ldots x_{m_{kj}^j} \rightarrow P(f_{kj} x_1 \ldots x_{m_{kj}^j})))$$

where $t_j \text{.case}$ is the internal name of the case-construct. This theorem can be added to a simpset via addsplits (see §2.4.1).

Case splitting on assumption works as well, by using the rule $t_j \text{.split_asm}$ in the same manner. Both rules are available under $t_j \text{splits}$ (this name is not bound in ML, though).

By default only the selector expression ($e$ above) in a case-construct is simplified, in analogy with if (see page 22). Only if that reduces to a constructor is one of the arms of the case-construct exposed and simplified. To ensure full simplification of all parts of a case-construct for datatype $t$, remove $t \text{.case_weak_cong}$ from the simpset, for example by delcongs [thm "t.case_cong_weak"].
The function size

Theory NatArith declares a generic function size of type $\alpha \Rightarrow \text{nat}$. Each datatype defines a particular instance of size by overloading according to the following scheme:

$$
\text{size}(C_i x_1 \ldots x_{m_i}) = \begin{cases} 
0 & \text{if } \text{Rec}_i = \emptyset \\
1 + \sum_{h=1}^{l_i} \text{size } x_{r_{i,h}} & \text{if } \text{Rec}_i = \{(r_{i,1,1}, s_{i,1}), \ldots, (r_{i,l_i,1}, s_{i,l_i})\}
\end{cases}
$$

where $\text{Rec}_i$ is defined above. Viewing datatypes as generalised trees, the size of a leaf is 0 and the size of a node is the sum of the sizes of its subtrees + 1.

2.6.2 Defining datatypes

The theory syntax for datatype definitions is given in the Isabelle/Isar reference manual. In order to be well-formed, a datatype definition has to obey the rules stated in the previous section. As a result the theory is extended with the new types, the constructors, and the theorems listed in the previous section.

Most of the theorems about datatypes become part of the default simpset and you never need to see them again because the simplifier applies them automatically. Only induction or case distinction are usually invoked by hand.

**induct_tac** "x" $i$ applies structural induction on variable $x$ to subgoal $i$, provided the type of $x$ is a datatype.

**induct_tac** "x1 ... xn" $i$ applies simultaneous structural induction on the variables $x_1, \ldots, x_n$ to subgoal $i$. This is the canonical way to prove properties of mutually recursive datatypes such as aexp and bexp, or datatypes with nested recursion such as term.

In some cases, induction is overkill and a case distinction over all constructors of the datatype suffices.

**case_tac** "u" $i$ performs a case analysis for the term $u$ whose type must be a datatype. If the datatype has $k_j$ constructors $C_{i_1}^j, \ldots, C_{i_{k_j}}^j$, subgoal $i$ is replaced by $k_j$ new subgoals which contain the additional assumption $u = C_{i_i}^j x_1 \ldots x_{m_i}$ for $i_i = 1, \ldots, k_j$.

Note that induction is only allowed on free variables that should not occur among the premises of the subgoal. Case distinction applies to arbitrary terms.
For the technically minded, we exhibit some more details. Processing the theory file produces an ML structure which, in addition to the usual components, contains a structure named $t$ for each datatype $t$ defined in the file. Each structure $t$ contains the following elements:

\begin{verbatim}
val distinct : thm list
val inject : thm list
val induct : thm
val exhaust : thm
val cases : thm list
val split : thm
val split_asm : thm
val recs : thm list
val size : thm list
val simps : thm list
\end{verbatim}

distinct, inject, induct, size and split contain the theorems described above. For user convenience, distinct contains inequalities in both directions. The reduction rules of the case-construct are in cases. All theorems from distinct, inject and cases are combined in simps. In case of mutually recursive datatypes, recs, size, induct and simps are contained in a separate structure named $t_1\ldots t_n$.

### 2.7 Old-style recursive function definitions

Old-style recursive definitions via recdef requires that you supply a well-founded relation that governs the recursion. Recursive calls are only allowed if they make the argument decrease under the relation. Complicated recursion forms, such as nested recursion, can be dealt with. Termination can even be proved at a later time, though having unsolved termination conditions around can make work difficult.\footnote{This facility is based on Konrad Slind’s TFL package [16]. Thanks are due to Konrad for implementing TFL and assisting with its installation.}

Using recdef, you can declare functions involving nested recursion and pattern-matching. Recursion need not involve datatypes and there are few syntactic restrictions. Termination is proved by showing that each recursive call makes the argument smaller in a suitable sense, which you specify by supplying a well-founded relation.

Here is a simple example, the Fibonacci function. The first line declares fib to be a constant. The well-founded relation is simply $<$ (on the natural numbers). Pattern-matching is used here: 1 is a macro for Suc 0.
consts fib :: "nat => nat"
recdef fib "less_than"
  "fib 0 = 0"
  "fib 1 = 1"
  "fib (Suc(Suc x)) = (fib x + fib (Suc x))"

With recdef, function definitions may be incomplete, and patterns may overlap, as in functional programming. The recdef package disambiguates overlapping patterns by taking the order of rules into account. For missing patterns, the function is defined to return a default value.

The well-founded relation defines a notion of “smaller” for the function’s argument type. The relation < is well-founded provided it admits no infinitely decreasing chains

\[ \cdots \prec x_n \prec \cdots \prec x_1. \]

If the function’s argument has type \( \tau \), then \( \prec \) has to be a relation over \( \tau \): it must have type \((\tau \times \tau)\)set.

Proving well-foundedness can be tricky, so Isabelle/HOL provides a collection of operators for building well-founded relations. The package recognises these operators and automatically proves that the constructed relation is well-founded. Here are those operators, in order of importance:

- **less_than** is “less than” on the natural numbers. (It has type \((\text{nat} \times \text{nat})\)set, while < has type \([\text{nat}, \text{nat}] \Rightarrow \text{bool}\).)

- **measure** \(f\), where \(f\) has type \(\tau \Rightarrow \text{nat}\), is the relation < on type \(\tau\) such that \(x \prec y\) if and only if \(f(x) < f(y)\). Typically, \(f\) takes the recursive function’s arguments (as a tuple) and returns a result expressed in terms of the function size. It is called a measure function. Recall that size is overloaded and is defined on all datatypes (see §2.6.1).

- **inv_image** \(R f\) is a generalisation of measure. It specifies a relation such that \(x \prec y\) if and only if \(f(x)\) is less than \(f(y)\) according to \(R\), which must itself be a well-founded relation.

- \(R_1<\text{lex}*>R_2\) is the lexicographic product of two relations. It is a relation on pairs and satisfies \((x_1, x_2) \prec (y_1, y_2)\) if and only if \(x_1\) is less than \(y_1\) according to \(R_1\) or \(x_1 = y_1\) and \(x_2\) is less than \(y_2\) according to \(R_2\).

- **finite_psubset** is the proper subset relation on finite sets.

We can use measure to declare Euclid’s algorithm for the greatest common divisor. The measure function, \(\lambda(m, n). n\), specifies that the recursion terminates because argument \(n\) decreases.
CHAPTER 2. HIGHER-ORDER LOGIC

recedef gcd "measure ((%(m,n). n) ::nat*nat=>nat)"
  "gcd (m, n) = (if n=0 then m else gcd(n, m mod n))"

The general form of a well-founded recursive definition is

recedef function rel
  congs congruence rules (optional)
  simpset simplification set (optional)
  reduction rules

where

- **function** is the name of the function, either as an *id* or a *string*.

- **rel** is a HOL expression for the well-founded termination relation.

- **congruence rules** are required only in highly exceptional circumstances.

- The **simplification set** is used to prove that the supplied relation is well-founded. It is also used to prove the **termination conditions**: assertions that arguments of recursive calls decrease under **rel**. By default, simplification uses **simpset()**, which is sufficient to prove well-foundedness for the built-in relations listed above.

- **reduction rules** specify one or more recursion equations. Each left-hand side must have the form \( f\, t \), where \( f \) is the function and \( t \) is a tuple of distinct variables. If more than one equation is present then \( f \) is defined by pattern-matching on components of its argument whose type is a **datatype**.

  The ML identifier \( f\).simps contains the reduction rules as a list of theorems.

With the definition of \( \text{gcd} \) shown above, Isabelle/HOL is unable to prove one termination condition. It remains as a precondition of the recursion theorems:

\[
\text{gcd.simps;}
\text{"! m n. n \neq 0 \rightarrow m \mod n < n}
\rightarrow \text{gcd (?m,?n) = (if ?n=0 then ?m else gcd (?n, ?m mod ?n))"}]
\text{: thm list}

The theory **HOL/ex/Primes** illustrates how to prove termination conditions afterwards. The function **Tfl.tgoalw** is like the standard function **goalw**, which sets up a goal to prove, but its argument should be the identifier \( f\).simps and its effect is to set up a proof of the termination conditions:
This subgoal has a one-step proof using simp_tac. Once the theorem is proved, it can be used to eliminate the termination conditions from elements of gcd.simps. Theory HOL/Subst/Unify is a much more complicated example of this process, where the termination conditions can only be proved by complicated reasoning involving the recursive function itself.

Isabelle/HOL can prove the gcd function’s termination condition automatically if supplied with the right simpset.

If all termination conditions were proved automatically, f.simps is added to the simpset automatically, just as in primrec. The simplification rules corresponding to clause i (where counting starts at 0) are called f.i and can be accessed as thms "f.i", which returns a list of theorems. Thus you can, for example, remove specific clauses from the simpset. Note that a single clause may give rise to a set of simplification rules in order to capture the fact that if clauses overlap, their order disambiguates them.

A recdef definition also returns an induction rule specialised for the recursive function. For the gcd function above, the induction rule is

This rule should be used to reason inductively about the gcd function. It usually makes the induction hypothesis available at all recursive calls, leading to very direct proofs. If any termination conditions remain unproved, they will become additional premises of this rule.

### 2.8 Example: Cantor’s Theorem

Cantor’s Theorem states that every set has more subsets than it has elements. It has become a favourite example in higher-order logic since it is so easily expressed:

\[ \forall f :: \alpha \Rightarrow bool . \exists S :: \alpha \Rightarrow bool . \forall x :: \alpha . f x \neq S \]
Viewing types as sets, $\alpha \Rightarrow \text{bool}$ represents the powerset of $\alpha$. This version states that for every function from $\alpha$ to its powerset, some subset is outside its range.

The Isabelle proof uses HOL’s set theory, with the type $\alpha \text{ set}$ and the operator \texttt{range}.

context Set.thy;

The set $S$ is given as an unknown instead of a quantified variable so that we may inspect the subset found by the proof.

Goal "?S ~: range (f :: 'a=>'a set)"
Level 0
?S ~: range f
1. ?S ~: range f

The first two steps are routine. The rule \texttt{rangeE} replaces $?S \in \text{range } f$ by $?S = f x$ for some $x$.

by (resolve_tac [notI] 1);
Level 1
?S ~: range f
1. ?S : range f ==> False
by (eresolve_tac [rangeE] 1);
Level 2
?S ~: range f
1. !!x. ?S = f x ==> False

Next, we apply \texttt{equalityCE}, reasoning that since $?S = f x$, we have $?c \in ?S$ if and only if $?c \in f x$ for any $?c$.

by (eresolve_tac [equalityCE] 1);
Level 3
?S ~: range f
1. !!x. [| ?c3 x : ?S; ?c3 x : f x |] ==> False
2. !!x. [| ?c3 x ~: ?S; ?c3 x ~: f x |] ==> False

Now we use a bit of creativity. Suppose that $?S$ has the form of a comprehension. Then $?c \in \{ x . ?P x \}$ implies $?P ?c$. Destruct-resolution using \texttt{CollectD} instantiates $?S$ and creates the new assumption.

by (dresolve_tac [CollectD] 1);
Level 4
{x. ?P7 x} ~: range f
1. !!x. [| ?c3 x : f x; ?P7(?c3 x) |] ==> False
2. !!x. [| ?c3 x ~: {x. ?P7 x}; ?c3 x ~: f x |] ==> False

Forcing a contradiction between the two assumptions of subgoal 1 completes the instantiation of $S$. It is now the set $\{ x . x \not\in f x \}$, which is the standard diagonal construction.
by (contr_tac 1);
Level 5
{x. x ~: f x} ~: range f
1. !!x. [| x ~: {x. x ~: f x}; x ~: f x |] ==> False

The rest should be easy. To apply CollectI to the negated assumption, we employ swap_res_tac:

by (swap_res_tac [CollectI] 1);
Level 6
{x. x ~: f x} ~: range f
1. !!x. [| x ~: f x; ~ False |] ==> x ~: f x
by (assume_tac 1);
Level 7
{x. x ~: f x} ~: range f
No subgoals!

How much creativity is required? As it happens, Isabelle can prove this theorem automatically. The default classical set claset() contains rules for most of the constructs of HOL’s set theory. We must augment it with equalityCE to break up set equalities, and then apply best-first search. Depth-first search would diverge, but best-first search successfully navigates through the large search space.

choplev 0;
Level 0
?S ~: range f
1. ?S ~: range f
by (best_tac (claset() addSEs [equalityCE]) 1);
Level 1
{x. x ~: f x} ~: range f
No subgoals!

If you run this example interactively, make sure your current theory contains theory Set, for example by executing context Set.thy. Otherwise the default claset may not contain the rules for set theory.
First-Order Sequent Calculus

The theory LK implements classical first-order logic through Gentzen’s sequent calculus (see Gallier [5] or Takeuti [17]). Resembling the method of semantic tableaux, the calculus is well suited for backwards proof. Assertions have the form \( \Gamma \vdash \Delta \), where \( \Gamma \) and \( \Delta \) are lists of formulae. Associative unification, simulated by higher-order unification, handles lists (§3.7 presents details, if you are interested).

The logic is many-sorted, using Isabelle’s type classes. The class of first-order terms is called \texttt{term}\. No types of individuals are provided, but extensions can define types such as \texttt{nat::term} and type constructors such as \texttt{list::(term)term}. Below, the type variable \( \alpha \) ranges over class \texttt{term}; the equality symbol and quantifiers are polymorphic (many-sorted). The type of formulae is \texttt{o}, which belongs to class \texttt{logic}.

LK implements a classical logic theorem prover that is nearly as powerful as the generic classical reasoner. The simplifier is now available too.

To work in LK, start up Isabelle specifying \texttt{Sequents} as the object-logic. Once in Isabelle, change the context to theory \texttt{LK.thy}:

\begin{verbatim}
isabelle Sequents
context LK.thy;
\end{verbatim}

Modal logic and linear logic are also available, but unfortunately they are not documented.

3.1 Syntax and rules of inference

Figure 3.1 gives the syntax for LK, which is complicated by the representation of sequents. Type \( \texttt{sobj \Rightarrow sobj} \) represents a list of formulae.

The \textbf{definite description} operator \( \iota x \cdot P[x] \) stands for some \( a \) satisfying \( P[a] \), if one exists and is unique. Since all terms in LK denote something, a description is always meaningful, but we do not know its value unless \( P[x] \) defines it uniquely. The Isabelle notation is \texttt{THE x. P[x]}. The corresponding rule (Fig. 3.4) does not entail the Axiom of Choice because it requires uniqueness.
## CHAPTER 3. FIRST-ORDER SEQUENT CALCULUS

### Constants

<table>
<thead>
<tr>
<th>symbol</th>
<th>name</th>
<th>meta-type</th>
<th>priority</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL</td>
<td>All</td>
<td>(α ⇒ o) ⇒ o</td>
<td>10</td>
<td>universal quantifier (∀)</td>
</tr>
<tr>
<td>EX</td>
<td>Ex</td>
<td>(α ⇒ o) ⇒ o</td>
<td>10</td>
<td>existential quantifier (∃)</td>
</tr>
<tr>
<td>THE</td>
<td>The</td>
<td>(α ⇒ o) ⇒ α</td>
<td>10</td>
<td>definite description (ι)</td>
</tr>
</tbody>
</table>

### Binders

<table>
<thead>
<tr>
<th>symbol</th>
<th>meta-type</th>
<th>priority</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>[α, α] ⇒ o</td>
<td>Left 50</td>
<td>equality (=)</td>
</tr>
<tr>
<td>&amp;</td>
<td>[o, o] ⇒ o</td>
<td>Right 35</td>
<td>conjunction (∧)</td>
</tr>
<tr>
<td></td>
<td>[o, o] ⇒ o</td>
<td>Right 30</td>
<td>disjunction (∨)</td>
</tr>
<tr>
<td>---</td>
<td>[o, o] ⇒ o</td>
<td>Right 25</td>
<td>implication (→)</td>
</tr>
<tr>
<td>&lt;-&gt;</td>
<td>[o, o] ⇒ o</td>
<td>Right 25</td>
<td>biconditional (↔)</td>
</tr>
</tbody>
</table>

### Infixes

<table>
<thead>
<tr>
<th>external</th>
<th>internal</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Γ ⊢ Δ</td>
<td>Trueprop(Γ, Δ)</td>
<td>sequent Γ ⊢ Δ</td>
</tr>
</tbody>
</table>

### Translations

Figure 3.1: Syntax of LK
prop = sequence |- sequence

sequence = elem (, elem)*
| empty

elem = $ term
| formula
| <<sequence>>

formula = expression of type o
| term = term
| ~ formula
| formula & formula
| formula | formula
| formula --> formula
| formula <-> formula
| ALL id id . formula
| EX id id . formula
| THE id . formula

Figure 3.2: Grammar of LK
CHAPTER 3. FIRST-ORDER SEQUENT CALCULUS

\begin{itemize}
  \item basic \quad \frac{H, P, G \vdash E, P, F}{H, P, G \vdash E, P, F}
  \item contRS \quad \frac{H \vdash E, S, S, S \vdash F \Rightarrow H \vdash E, S, F}{H \vdash E, S, F}
  \item contLS \quad \frac{H, S, S, G \vdash E}{H, S, G \vdash E}
  \item thinRS \quad \frac{H \vdash E, F \Rightarrow H \vdash E, S, F}{H \vdash E, F}
  \item thinLS \quad \frac{H, G \vdash E}{H, S, G \vdash E}
  \item cut \quad \frac{[H \vdash E, P; H, P \vdash E]}{H \vdash E}
\end{itemize}

\textbf{Structural rules}

\begin{itemize}
  \item refl \quad \frac{H \vdash E, a=a, F}{H \vdash E, a=a, F}
  \item subst \quad \frac{H(a), G(a) \vdash E(a)}{H(b), a=b, G(b) \vdash E(b)}
\end{itemize}

\textbf{Equality rules}

Figure 3.2 presents the grammar of LK. Traditionally, $\Gamma$ and $\Delta$ are metavariables for sequences. In Isabelle's notation, the prefix $\$ on a term makes it range over sequences. In a sequent, anything not prefixed by $\$ is taken as a formula.

The notation $<<\text{sequence}>>$ stands for a sequence of formulæ. For example, you can declare the constant \texttt{imps} to consist of two implications:

\begin{verbatim}
consts P,Q,R :: o
constdefs imps :: seq'\Rightarrow seq'
   "imps == <<P \Rightarrow Q, Q \Rightarrow R>>"
\end{verbatim}

Then you can use it in axioms and goals, for example

Conditional expressions are available with the notation

\begin{verbatim}
if formula then term else term.
\end{verbatim}

Figure 3.3: Basic Rules of LK
CHAPTER 3. FIRST-ORDER SEQUENT CALCULUS

48

True_def  True == False-->False
iff_def    P<->Q == (P-->Q) & (Q-->P)
conjR     [| $H|- $E, P, $F; $H|- $E, Q, $F |] ==> $H|- $E, P&Q, $F
conjL     $H, P, Q, $G |- $E ==> $H, P & Q, $G |- $E
disjR     $H |- $E, P, Q, $F ==> $H |- $E, P|Q, $F
impR      $H, P |- $E, Q, $F ==> $H |- $E, P-->Q, $F
notR      $H, P |- $E, $F ==> $H |- $E, ~P, $F
notL      $H, $G |- $E, P ==> $H, ~P, $G |- $E
FalseL    $H, False, $G |- $E

allR      (!!x. $H|- $E, P(x), $F) ==> $H|- $E, ALL x. P(x), $F
allL      $H, P(x), $G, ALL x. P(x) |- $E ==> $H, ALL x. P(x), $G|- $E
exR       $H|- $E, P(x), $F, EX x. P(x) ==> $H|- $E, EX x. P(x), $F
exL       (!!x. $H, P(x), $G|- $E) ==> $H, EX x. P(x), $G|- $E

The       [| $H |- $E, P(a), $F; !!x. $H, P(x) |- $E, x=a, $F |] ==> $H |- $E, P(THE x. P(x)), $F

Logical rules

Figure 3.4: Rules of LK
CHAPTER 3. FIRST-ORDER SEQUENT CALCULUS

49

Figure 3.5: Derived rules for LK

Goalw [imps_def] "P, $imps |- R";
Level 0
P, $imps |- R
1. P, P --> Q, Q --> R |- R
by (Fast_tac 1);
Level 1
P, $imps |- R
No subgoals!

Figures 3.3 and 3.4 present the rules of theory LK. The connective ↔ is
defined using ∧ and →. The axiom for basic sequents is expressed in a form
that provides automatic thinning: redundant formulae are simply ignored. The
other rules are expressed in the form most suitable for backward proof;
exchange and contraction rules are not normally required, although they are
provided anyway.

Figure 3.5 presents derived rules, including rules for ↔. The weakened
quantifier rules discard each quantification after a single use; in an automatic
proof procedure, they guarantee termination, but are incomplete. Multiple use of a quantifier can be obtained by a contraction rule, which in backward proof duplicates a formula. The tactic \texttt{res_inst_tac} can instantiate the variable \texttt{?P} in these rules, specifying the formula to duplicate. See theory \texttt{Sequents/LK0} in the sources for complete listings of the rules and derived rules.

To support the simplifier, hundreds of equivalences are proved for the logical connectives and for if-then-else expressions. See the file \texttt{Sequents/simpdata.ML}.

### 3.2 Automatic Proof

LK instantiates Isabelle’s simplifier. Both equality (=) and the biconditional (\(\leftrightarrow\)) may be used for rewriting. The tactic \texttt{Simp_tac} refers to the default simpset (\texttt{simpset()}). With sequents, the \texttt{full} and \texttt{asm} forms of the simplifier are not required; all the formulae in the sequent will be simplified. The left-hand formulae are taken as rewrite rules. (Thus, the behaviour is what you would normally expect from calling \texttt{Asm_full_simp_tac}.)

For classical reasoning, several tactics are available:

- \texttt{Safe_tac : int \rightarrow tactic}
- \texttt{Step_tac : int \rightarrow tactic}
- \texttt{Fast_tac : int \rightarrow tactic}
- \texttt{Best_tac : int \rightarrow tactic}
- \texttt{Pc_tac : int \rightarrow tactic}

These refer not to the standard classical reasoner but to a separate one provided for the sequent calculus. Two commands are available for adding new sequent calculus rules, safe or unsafe, to the default “theorem pack”:

- \texttt{Add_safes : thm list \rightarrow unit}
- \texttt{Add_unsafes : thm list \rightarrow unit}

To control the set of rules for individual invocations, lower-case versions of all these primitives are available. Sections 3.8 and 3.9 give full details.

### 3.3 Tactics for the cut rule

According to the cut-elimination theorem, the cut rule can be eliminated from proofs of sequents. But the rule is still essential. It can be used to structure a proof into lemmas, avoiding repeated proofs of the same formula. More importantly, the cut rule cannot be eliminated from derivations of rules.
For example, there is a trivial cut-free proof of the sequent $P \land Q \vdash Q \land P$. Noting this, we might want to derive a rule for swapping the conjuncts in a right-hand formula:

$$
\Gamma \vdash \Delta, P \land Q \\
\Gamma \vdash \Delta, Q \land P
$$

The cut rule must be used, for $P \land Q$ is not a subformula of $Q \land P$. Most cuts directly involve a premise of the rule being derived (a meta-assumption). In a few cases, the cut formula is not part of any premise, but serves as a bridge between the premises and the conclusion. In such proofs, the cut formula is specified by calling an appropriate tactic.

```ml
let cutR_tac : string -> int -> tactic = 
let cutL_tac : string -> int -> tactic = 
```

These tactics refine a subgoal into two by applying the cut rule. The cut formula is given as a string, and replaces some other formula in the sequent.

```ml
let cutR_tac P i = 
let cutL_tac P i = 
```

All the structural rules — cut, contraction, and thinning — can be applied to particular formulae using `res_inst_tac`.

### 3.4 Tactics for sequents

```ml
let forms_of_seq : term -> term list = 
let could_res : term * term -> bool = 
let could_resolve_seq : term * term -> bool = 
let filseq_resolve_tac : thm list -> int -> int -> tactic = 
```

Associative unification is not as efficient as it might be, in part because the representation of lists defeats some of Isabelle’s internal optimisations. The following operations implement faster rule application, and may have other uses.

```ml
let forms_of_seq t = 
```
could_res \( (t,u) \) tests whether two formula lists could be resolved. List \( t \) is from a premise or subgoal, while \( u \) is from the conclusion of an object-rule. Assuming that each formula in \( u \) is surrounded by sequence variables, it checks that each conclusion formula is unifiable (using could_unify) with some subgoal formula.

could_resolve_seq \( (t,u) \) tests whether two sequents could be resolved. Sequent \( t \) is a premise or subgoal, while \( u \) is the conclusion of an object-rule. It simply calls could_res twice to check that both the left and the right sides of the sequents are compatible.

filseq_resolve_tac thms maxr i uses filter_thms could_resolve to extract the \( thms \) that are applicable to subgoal \( i \). If more than \( maxr \) theorems are applicable then the tactic fails. Otherwise it calls resolve_tac. Thus, it is the sequent calculus analogue of filt_resolve_tac.

### 3.5 A simple example of classical reasoning

The theorem \( \vdash \exists y. \forall x. P(y) \rightarrow P(x) \) is a standard example of the classical treatment of the existential quantifier. Classical reasoning is easy using LK, as you can see by comparing this proof with the one given in the FOL manual [12]. From a logical point of view, the proofs are essentially the same; the key step here is to use \texttt{exR} rather than the weaker \texttt{exR_thin}.

Goal "\( |- \exists y. \forall x. P(y) \rightarrow P(x) \)"

Level 0
\( |- \exists y. \forall x. P(y) \rightarrow P(x) \)
1. \( |- \exists y. \forall x. P(y) \rightarrow P(x) \)
   by (resolve_tac [exR] 1);

Level 1
\( |- \exists y. \forall x. P(y) \rightarrow P(x) \)
1. \( |- \forall x. P(?x) \rightarrow P(x) \), \( \exists x. \forall xa. P(x) \rightarrow P(xa) \)
   by (resolve_tac [allR] 1);

Level 2
\( |- \forall x. P(?x) \rightarrow P(x) \)
1. \( \forall x. P(?x) \vdash P(x) \), \( \exists x. \forall xa. P(x) \rightarrow P(xa) \)
   by (resolve_tac [impR] 1);

Level 3
\( |- \exists x. \forall xa. P(x) \rightarrow P(xa) \)
1. \( \exists x. P(?x) \vdash P(x) \), \( \exists x. \forall xa. P(x) \rightarrow P(xa) \)

Because LK is a sequent calculus, the formula \( P(?x) \) does not become an
CHAPTER 3. FIRST-ORDER SEQUENT CALCULUS

assumption; instead, it moves to the left side. The resulting subgoal cannot be instantiated to a basic sequent: the bound variable $x$ is not unifiable with the unknown $\exists x$.

\[
\text{by (resolve_tac [basic] 1);}
\]
\[
\text{by: tactic failed}
\]

We reuse the existential formula using \texttt{exR\_thin}, which discards it; we shall not need it a third time. We again break down the resulting formula.

\[
\text{by (resolve_tac [exR\_thin] 1);} \\
\text{Level 4} \\
|- \exists y. \forall x. P(y) \rightarrow P(x) \\
1. \exists x. P(?x) |- P(x), \forall xa. P(?x7(x)) \rightarrow P(xa)
\]
\[
\text{by (resolve_tac [allR] 1);} \\
\text{Level 5} \\
|- \exists y. \forall x. P(y) \rightarrow P(x) \\
1. \exists x xa. P(?x) |- P(x), P(?x7(x)) \rightarrow P(xa)
\]
\[
\text{by (resolve_tac [impR] 1);} \\
\text{Level 6} \\
|- \exists y. \forall x. P(y) \rightarrow P(x) \\
1. \exists x xa. P(?x), P(?x7(x)) |- P(x), P(xa)
\]

Subgoal 1 seems to offer lots of possibilities. Actually the only useful step is instantiating $\exists x7$ to $\lambda x . x$, transforming $\exists x7(x)$ into $x$.

\[
\text{by (resolve_tac [basic] 1);} \\
\text{Level 7} \\
|- \exists y. \forall x. P(y) \rightarrow P(x) \\
\text{No subgoals!}
\]

This theorem can be proved automatically. Because it involves quantifier duplication, we employ best-first search:

\[
\text{Goal "|- EX y. ALL x. P(y)--P(x)";} \\
\text{Level 0} \\
|- EX y. ALL x. P(y) \rightarrow P(x) \\
1. |- EX y. ALL x. P(y) \rightarrow P(x) \\
\text{by (best_tac LK\_dup\_pack 1);} \\
\text{Level 1} \\
|- EX y. ALL x. P(y) \rightarrow P(x) \\
\text{No subgoals!}
\]

3.6 A more complex proof

Many of Pelletier’s test problems for theorem provers [15] can be solved automatically. Problem 39 concerns set theory, asserting that there is no Russell
CHAPTER 3. FIRST-ORDER SEQUENT CALCULUS

set — a set consisting of those sets that are not members of themselves:

\[ \vdash \neg (\exists x. \forall y. y \in x \leftrightarrow y \not\in y) \]

This does not require special properties of membership; we may generalize \( x \in y \) to an arbitrary predicate \( F(x, y) \). The theorem, which is trivial for \texttt{Fast_tac}, has a short manual proof. See the directory \\texttt{Sequents/LK} for many more examples.

We set the main goal and move the negated formula to the left.

Goal "\( \vdash \neg (\exists x. \forall y. F(y,x) \leftrightarrow \neg F(y,y)) \)"

Level 0

\( \vdash \neg (\exists x. \forall y. F(y,x) \leftrightarrow \neg F(y,y)) \)

1. \( \vdash \neg (\exists x. \forall y. F(y,x) \leftrightarrow \neg F(y,y)) \)

by (resolve_tac [notR] 1);

Level 1

\( \vdash \neg (\exists x. \forall y. F(y,x) \leftrightarrow \neg F(y,y)) \)

1. \( \exists x. \forall y. F(y,x) \leftrightarrow \neg F(y,y) \)

by (resolve_tac [exL] 1);

Level 2

\( \vdash \neg (\exists x. \forall y. F(y,x) \leftrightarrow \neg F(y,y)) \)

1. \( \forall x. \forall y. F(y,x) \leftrightarrow \neg F(y,y) \)

by (resolve_tac [allL_thin] 1);

Level 3

\( \vdash \neg (\exists x. \forall y. F(y,x) \leftrightarrow \neg F(y,y)) \)

1. \( \forall x. F(?x2(x),x) \leftrightarrow \neg F(?x2(x),?x2(x)) \)

The right side is empty; we strip both quantifiers from the formula on the left.

by (resolve_tac [exL] 1);

Level 2

\( \vdash \neg (\exists x. \forall y. F(y,x) \leftrightarrow \neg F(y,y)) \)

1. \( \forall x. F(?x2(x),x) \leftrightarrow \neg F(?x2(x),?x2(x)) \)

by (resolve_tac [allL_thin] 1);

Level 3

\( \vdash \neg (\exists x. \forall y. F(y,x) \leftrightarrow \neg F(y,y)) \)

1. \( \forall x. F(?x2(x),x) \leftrightarrow \neg F(?x2(x),?x2(x)) \)

The rule \texttt{iffL} says, if \( P \leftrightarrow Q \) then \( P \) and \( Q \) are either both true or both false. It yields two subgoals.

by (resolve_tac [iffL] 1);

Level 4

\( \vdash \neg (\exists x. \forall y. F(y,x) \leftrightarrow \neg F(y,y)) \)

1. \( \forall x. F(?x2(x),x) \leftrightarrow \neg F(?x2(x),?x2(x)) \)

2. \( \forall x. \neg F(?x2(x),?x2(x)) \rightarrow F(?x2(x),x) \)

We must instantiate \(?x2\), the shared unknown, to satisfy both subgoals. Beginning with subgoal 2, we move a negated formula to the left and create a basic sequent.
by (resolve_tac [notL] 2);
Level 5
\[ \vdash \neg (\exists x. \forall y. F(y,x) \leftrightarrow \neg F(y,y)) \]
1. \( \forall x. \vdash F(?x2(x),x) \rightarrow F(?x2(x),?x2(x)) \)
2. \( \forall x. \vdash F(?x2(x),x) \rightarrow F(?x2(x),?x2(x)) \)
by (resolve_tac [basic] 2);
Level 6
\[ \vdash \neg (\exists x. \forall y. F(y,x) \leftrightarrow \neg F(y,y)) \]
1. \( \forall x. \vdash F(x,x) \rightarrow F(x,x) \)
by (resolve_tac [basic] 1);
Level 7
\[ \vdash \neg (\exists x. \forall y. F(y,x) \leftrightarrow \neg F(y,y)) \]
1. \( \forall x. \vdash F(x,x) \rightarrow F(x,x) \)
Thanks to the instantiation of \(?x2\), subgoal 1 is obviously true.

by (resolve_tac [notR] 1);
Level 7
\[ \vdash \neg (\exists x. \forall y. F(y,x) \leftrightarrow \neg F(y,y)) \]
1. \( \forall x. \vdash F(x,x) \rightarrow F(x,x) \)
by (resolve_tac [basic] 1);
Level 8
\[ \vdash \neg (\exists x. \forall y. F(y,x) \leftrightarrow \neg F(y,y)) \]
No subgoals!

3.7 *Unification for lists

Higher-order unification includes associative unification as a special case, by an encoding that involves function composition [7, page 37]. To represent lists, let \( C \) be a new constant. The empty list is \( \lambda x. x \), while \([t_1, t_2, \ldots, t_n] \) is represented by
\[ \lambda x. C(t_1, C(t_2, \ldots, C(t_n, x))). \]
The unifiers of this with \( \lambda x. \ y(x) \) give all the ways of expressing \([t_1, t_2, \ldots, t_n] \) as the concatenation of two lists.

Unlike orthodox associative unification, this technique can represent certain infinite sets of unifiers by flex-flex equations. But note that the term \( \lambda x. C(t, ?a) \) does not represent any list. Flex-flex constraints containing such garbage terms may accumulate during a proof.

This technique lets Isabelle formalize sequent calculus rules, where the comma is the associative operator:
\[
\frac{\Gamma, P, Q, \Delta \vdash \Theta}{\Gamma, P \land Q, \Delta \vdash \Theta} \quad (\land\text{-left})
\]
Multiple unifiers occur whenever this is resolved against a goal containing more than one conjunction on the left.

LK exploits this representation of lists. As an alternative, the sequent calculus can be formalized using an ordinary representation of lists, with a
logic program for removing a formula from a list. Amy Felty has applied this technique using the language λProlog [4].

Explicit formalization of sequents can be tiresome. But it gives precise control over contraction and weakening, and is essential to handle relevant and linear logics.

3.8 *Packaging sequent rules

The sequent calculi come with simple proof procedures. These are incomplete but are reasonably powerful for interactive use. They expect rules to be classified as **safe** or **unsafe**. A rule is safe if applying it to a provable goal always yields provable subgoals. If a rule is safe then it can be applied automatically to a goal without destroying our chances of finding a proof. For instance, all the standard rules of the classical sequent calculus LK are safe. An unsafe rule may render the goal unprovable; typical examples are the weakened quantifier rules \text{allL\_thin} and \text{exR\_thin}.

Proof procedures use safe rules whenever possible, using an unsafe rule as a last resort. Those safe rules are preferred that generate the fewest subgoals. Safe rules are (by definition) deterministic, while the unsafe rules require a search strategy, such as backtracking.

A **pack** is a pair whose first component is a list of safe rules and whose second is a list of unsafe rules. Packs can be extended in an obvious way to allow reasoning with various collections of rules. For clarity, LK declares **pack** as an ML datatype, although is essentially a type synonym:

```
datatype pack = Pack of thm list * thm list;
```

Pattern-matching using constructor **Pack** can inspect a pack’s contents. Packs support the following operations:

```
pack         : unit -> pack
pack_of      : theory -> pack
empty_pack   : pack
prop_pack    : pack
LK_pack      : pack
LK_dup_pack  : pack
add_safes    : pack * thm list -> pack                     infix 4
add_unsafes  : pack * thm list -> pack                     infix 4
```

**pack** returns the pack attached to the current theory.

**pack\_of thy** returns the pack attached to theory **thy**.

**empty\_pack** is the empty pack.
\textbf{CHAPTER 3. FIRST-ORDER SEQUENT CALCULUS}

\texttt{prop\_pack} contains the propositional rules, namely those for $\land$, $\lor$, $\neg$, $\rightarrow$ and $\leftrightarrow$, along with the rules \texttt{basic} and \texttt{refl}. These are all safe.

\texttt{LK\_pack} extends \texttt{prop\_pack} with the safe rules \texttt{allR} and \texttt{exL} and the unsafe rules \texttt{allL\_thin} and \texttt{exR\_thin}. Search using this is incomplete since quantified formulae are used at most once.

\texttt{LK\_dup\_pack} extends \texttt{prop\_pack} with the safe rules \texttt{allR} and \texttt{exL} and the unsafe rules \texttt{allL} and \texttt{exR}. Search using this is complete, since quantified formulae may be reused, but frequently fails to terminate. It is generally unsuitable for depth-first search.

\texttt{pack add\_safes rules} adds some safe rules to the pack \texttt{pack}.

\texttt{pack add\_unsafes rules} adds some unsafe rules to the pack \texttt{pack}.

### 3.9 *Proof procedures*

The LK proof procedure is similar to the classical reasoner described in the Reference Manual. In fact it is simpler, since it works directly with sequents rather than simulating them. There is no need to distinguish introduction rules from elimination rules, and of course there is no swap rule. As always, Isabelle’s classical proof procedures are less powerful than resolution theorem provers. But they are more natural and flexible, working with an open-ended set of rules.

Backtracking over the choice of a safe rule accomplishes nothing: applying them in any order leads to essentially the same result. Backtracking may be necessary over basic sequents when they perform unification. Suppose that 0, 1, 2, 3 are constants in the subgoals

\[
P(0), P(1), P(2) \vdash P(\textit{?a})
\]

\[
P(0), P(2), P(3) \vdash P(\textit{?a})
\]

\[
P(1), P(3), P(2) \vdash P(\textit{?a})
\]

The only assignment that satisfies all three subgoals is $\textit{?a} \mapsto 2$, and this can only be discovered by search. The tactics given below permit backtracking only over axioms, such as \texttt{basic} and \texttt{refl}; otherwise they are deterministic.

#### 3.9.1 Method A

\begin{verbatim}
  reresolve_tac : thm list -> int -> tactic
  repeat_goal_tac : pack -> int -> tactic
  pc_tac : pack -> int -> tactic
\end{verbatim}

These tactics use a method developed by Philippe de Groote. A subgoal
is refined and the resulting subgoals are attempted in reverse order. For some reason, this is much faster than attempting the subgoals in order. The method is inherently depth-first.

At present, these tactics only work for rules that have no more than two premises. They fail — return no next state — if they can do nothing.

\[ \text{reresolve_tac thms i} \]
repeatedly applies the \( \text{thms} \) to subgoal \( i \) and the resulting subgoals.

\[ \text{repeat_goal_tac pack i} \]
applies the safe rules in the pack to a goal and the resulting subgoals. If no safe rule is applicable then it applies an unsafe rule and continues.

\[ \text{pc_tac pack i} \]
applies \( \text{repeat_goal_tac} \) using depth-first search to solve subgoal \( i \).

### 3.9.2 Method B

\[ \text{safe_tac : pack -> int -> tactic} \]
\[ \text{step_tac : pack -> int -> tactic} \]
\[ \text{fast_tac : pack -> int -> tactic} \]
\[ \text{best_tac : pack -> int -> tactic} \]

These tactics are analogous to those of the generic classical reasoner. They use ‘Method A’ only on safe rules. They fail if they can do nothing.

\[ \text{safe_goal_tac pack i} \]
applies the safe rules in the pack to a goal and the resulting subgoals. It ignores the unsafe rules.

\[ \text{step_tac pack i} \]
either applies safe rules (using \( \text{safe_goal_tac} \)) or applies one unsafe rule.

\[ \text{fast_tac pack i} \]
applies \( \text{step_tac} \) using depth-first search to solve subgoal \( i \). Despite its name, it is frequently slower than \( \text{pc_tac} \).

\[ \text{best_tac pack i} \]
applies \( \text{step_tac} \) using best-first search to solve subgoal \( i \). It is particularly useful for quantifier duplication (using \( \text{LK_dup_pack} \)).
Chapter 4

Defining A Sequent-Based Logic

The Isabelle theory Sequents.thy provides facilities for using sequent notation in users’ object logics. This theory allows users to easily interface the surface syntax of sequences with an underlying representation suitable for higher-order unification.

4.1 Concrete syntax of sequences

Mathematicians and logicians have used sequences in an informal way much before proof systems such as Isabelle were created. It seems sensible to allow people using Isabelle to express sequents and perform proofs in this same informal way, and without requiring the theory developer to spend a lot of time in ML programming.

By using Sequents.thy appropriately, a logic developer can allow users to refer to sequences in several ways:

- A sequence variable is any alphanumeric string with the first character being a $ sign. So, consider the sequent $A \vdash B$, where $A$ is intended to match a sequence of zero or more items.

- A sequence with unspecified sub-sequences and unspecified or individual items is written as a comma-separated list of regular variables (representing items), particular items, and sequence variables, as in

  $A, B, C, D(x) \vdash E$

  Here both $A$ and $D(x)$ are allowed to match any subsequences of items on either side of the two items that match $B$ and $C$. Moreover, the sequence matching $D(x)$ may contain occurrences of $x$.

- An empty sequence can be represented by a blank space, as in $\vdash \text{true}$.
These syntactic constructs need to be assimilated into the object theory being developed. The type that we use for these visible objects is given the name \texttt{seq}. A \texttt{seq} is created either by the empty space, a \texttt{seqobj} or a \texttt{seqobj} followed by a \texttt{seq}, with a comma between them. A \texttt{seqobj} is either an item or a variable representing a sequence. Thus, a theory designer can specify a function that takes two sequences and returns a meta-level proposition by giving it the Isabelle type \texttt{[seq, seq] => prop}.

This is all part of the concrete syntax, but one may wish to exploit Isabelle’s higher-order abstract syntax by actually having a different, more powerful \textit{internal} syntax.

### 4.2 Basis

One could opt to represent sequences as first-order objects (such as simple lists), but this would not allow us to use many facilities Isabelle provides for matching. By using a slightly more complex representation, users of the logic can reap many benefits in facilities for proofs and ease of reading logical terms.

A sequence can be represented as a function — a constructor for further sequences — by defining a binary \textit{abstract} function \texttt{Seq0'} with type \texttt{[o, seq']=>seq'}, and translating a sequence such as \texttt{A, B, C} into

\[
\% s. \texttt{Seq0'}(A, \texttt{Seq0'}(B, \texttt{Seq0'}(C, s)))
\]

This sequence can therefore be seen as a constructor for further sequences. The constructor \texttt{Seq0'} is never given a value, and therefore it is not possible to evaluate this expression into a basic value.

Furthermore, if we want to represent the sequence \texttt{A, B, C}, we note that \texttt{B} already represents a sequence, so we can use \texttt{B} itself to refer to the function, and therefore the sequence can be mapped to the internal form:

\[
\% s. \texttt{Seq0'}(A, \texttt{B}(\texttt{Seq0'}(C, s)))
\]

So, while we wish to continue with the standard, well-liked \textit{external} representation of sequences, we can represent them \textit{internally} as functions of type \texttt{seq' => seq'}.

### 4.3 Object logics

Recall that object logics are defined by mapping elements of particular types to the Isabelle type \texttt{prop}, usually with a function called \texttt{Trueprop}. So, an object logic proposition \texttt{P} is matched to the Isabelle proposition \texttt{Trueprop(P)}.
The name of the function is often hidden, so the user just sees \( P \). Isabelle is eager to make types match, so it inserts \texttt{Trueprop} automatically when an object of type \texttt{prop} is expected. This mechanism can be observed in most of the object logics which are direct descendants of \texttt{Pure}.

In order to provide the desired syntactic facilities for sequent calculi, rather than use just one function that maps object-level propositions to meta-level propositions, we use two functions, and separate internal from the external representation.

These functions need to be given a type that is appropriate for the particular form of sequents required: single or multiple conclusions. So multiple-conclusion sequents (used in the LK logic) can be specified by the following two definitions, which are lifted from the inbuilt \texttt{Sequents/LK.thy}:

\[
\texttt{Trueprop :: two_seqi} \\
\texttt{"@Trueprop" :: two_seqe ("(_,)/ - (_)") [6,6] 5}
\]

where the types used are defined in \texttt{Sequents.thy} as abbreviations:

\[
\texttt{two_seqi = [seq'=seq', seq'=seq'] => prop} \\
\texttt{two_seqe = [seq, seq] => prop}
\]

The next step is to actually create links into the low-level parsing and pretty-printing mechanisms, which map external and internal representations. These functions go below the user level and capture the underlying structure of Isabelle terms in \texttt{ml}. Fortunately the theory developer need not delve in this level; \texttt{Sequents.thy} provides the necessary facilities. All the theory developer needs to add in the \texttt{ml} section is a specification of the two translation functions:

\[
\texttt{ML} \\
\texttt{val parse_translation = \[(@Trueprop,Sequents.two_seq_tr "Trueprop")\];} \\
\texttt{val print_translation = \[("Trueprop",Sequents.two_seq_tr' @Trueprop")\];}
\]

In summary: in the logic theory being developed, the developer needs to specify the types for the internal and external representation of the sequences, and use the appropriate parsing and pretty-printing functions.

### 4.4 What’s in \texttt{Sequents.thy}

Theory \texttt{Sequents.thy} makes many declarations that you need to know about:

1. The Isabelle types given below, which can be used for the constants that map object-level sequents and meta-level propositions:
single_seqe = [seq, seqobj] => prop
single_seqi = [seq' => seq', seq' => seq'] => prop
two_seqi = [seq' => seq', seq' => seq'] => prop
two_seqe = [seq, seq] => prop
three_seqi = [seq' => seq', seq' => seq', seq' => seq'] => prop
three_seqe = [seq, seq, seq] => prop
four_seqi = [seq' => seq', seq' => seq', seq' => seq', seq' => seq'] => prop
four_seqe = [seq, seq, seq, seq] => prop

The single_ and two_ sets of mappings for internal and external representations are the ones used for, say single and multiple conclusion sequents. The other functions are provided to allow rules that manipulate more than two functions, as can be seen in the inbuilt object logics.

2. An auxiliary syntactic constant has been defined that directly maps a sequence to its internal representation:

"@Side" :: seq => (seq' => seq') ("<<(_)>>")

Whenever a sequence (such as "<< A, $B, $C>>") is entered using this syntax, it is translated into the appropriate internal representation. This form can be used only where a sequence is expected.

3. The ML functions single_tr, two_seq_tr, three_seq_tr, four_seq_tr for parsing, that is, the translation from external to internal form. Analogously there are single_tr', two_seq_tr', three_seq_tr', four_seq_tr' for pretty-printing, that is, the translation from internal to external form. These functions can be used in the ML section of a theory file to specify the translations to be used. As an example of use, note that in LK.thy we declare two identifiers:

val parse_translation = 
  [(@Trueprop, Sequents.two_seq_tr "Trueprop")];
val print_translation = 
  [("Trueprop", Sequents.two_seq_tr' "@Trueprop")];

The given parse translation will be applied whenever a @Trueprop constant is found, translating using two_seq_tr and inserting the constant Trueprop. The pretty-printing translation is applied analogously; a term that contains Trueprop is printed as a @Trueprop.
Constructive Type Theory

Martin-Löf’s Constructive Type Theory [9, 11] can be viewed at many different levels. It is a formal system that embodies the principles of intuitionistic mathematics; it embodies the interpretation of propositions as types; it is a vehicle for deriving programs from proofs.

Thompson’s book [18] gives a readable and thorough account of Type Theory. Nuprl is an elaborate implementation [3]. ALF is a more recent tool that allows proof terms to be edited directly [8].

Isabelle’s original formulation of Type Theory was a kind of sequent calculus, following Martin-Löf [9]. It included rules for building the context, namely variable bindings with their types. A typical judgement was

\[ a(x_1, \ldots, x_n) \in A(x_1, \ldots, x_n) \left[ x_1 \in A_1, x_2 \in A_2(x_1), \ldots, x_n \in A_n(x_1, \ldots, x_{n-1}) \right] \]

This sequent calculus was not satisfactory because assumptions like ‘suppose \( A \) is a type’ or ‘suppose \( B(x) \) is a type for all \( x \) in \( A \)’ could not be formalized.

The theory CTT implements Constructive Type Theory, using natural deduction. The judgement above is expressed using \( \land \) and \( \Rightarrow \):

\[ \land x_1, \ldots, x_n, [x_1 \in A_1; x_2 \in A_2(x_1); \ldots; x_n \in A_n(x_1, \ldots, x_{n-1})] \Rightarrow a(x_1, \ldots, x_n) \in A(x_1, \ldots, x_n) \]

Assumptions can use all the judgement forms, for instance to express that \( B \) is a family of types over \( A \):

\[ \land x. x \in A \Rightarrow B(x) \text{ type} \]

To justify the CTT formulation it is probably best to appeal directly to the semantic explanations of the rules [9], rather than to the rules themselves. The order of assumptions no longer matters, unlike in standard Type Theory. Contexts, which are typical of many modern type theories, are difficult to represent in Isabelle. In particular, it is difficult to enforce that all the variables in a context are distinct.

The theory does not use polymorphism. Terms in CTT have type \( i \), the type of individuals. Types in CTT have type \( t \).
<table>
<thead>
<tr>
<th>name</th>
<th>meta-type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>$t \rightarrow prop$</td>
<td>judgement form</td>
</tr>
<tr>
<td>Eqtype</td>
<td>$[t, t] \rightarrow prop$</td>
<td>judgement form</td>
</tr>
<tr>
<td>Elem</td>
<td>$[i, t] \rightarrow prop$</td>
<td>judgement form</td>
</tr>
<tr>
<td>Eqelem</td>
<td>$[i, i, t] \rightarrow prop$</td>
<td>judgement form</td>
</tr>
<tr>
<td>Reduce</td>
<td>$[i, i] \rightarrow prop$</td>
<td>extra judgement form</td>
</tr>
<tr>
<td>N</td>
<td>$t$</td>
<td>natural numbers type</td>
</tr>
<tr>
<td>0</td>
<td>$i$</td>
<td>constructor</td>
</tr>
<tr>
<td>succ</td>
<td>$i \rightarrow i$</td>
<td>constructor</td>
</tr>
<tr>
<td>rec</td>
<td>$[i, i, [i, i] \rightarrow i] \rightarrow i$</td>
<td>eliminator</td>
</tr>
<tr>
<td>Prod</td>
<td>$[t, i \rightarrow t] \rightarrow t$</td>
<td>general product type</td>
</tr>
<tr>
<td>lambda</td>
<td>$(i \rightarrow i) \rightarrow i$</td>
<td>constructor</td>
</tr>
<tr>
<td>Sum</td>
<td>$[t, i \rightarrow t] \rightarrow t$</td>
<td>general sum type</td>
</tr>
<tr>
<td>pair</td>
<td>$[i, i] \rightarrow i$</td>
<td>constructor</td>
</tr>
<tr>
<td>split</td>
<td>$[i, [i, i] \rightarrow i, i] \rightarrow i$</td>
<td>eliminator</td>
</tr>
<tr>
<td>fst snd</td>
<td>$i \rightarrow i$</td>
<td>projections</td>
</tr>
<tr>
<td>inl inr</td>
<td>$i \rightarrow i$</td>
<td>constructors for +</td>
</tr>
<tr>
<td>when</td>
<td>$[i, i \rightarrow i, i \rightarrow i] \rightarrow i$</td>
<td>eliminator for +</td>
</tr>
<tr>
<td>Eq</td>
<td>$[t, i, i] \rightarrow t$</td>
<td>equality type</td>
</tr>
<tr>
<td>eq</td>
<td>$i$</td>
<td>constructor</td>
</tr>
<tr>
<td>F</td>
<td>$t$</td>
<td>empty type</td>
</tr>
<tr>
<td>contr</td>
<td>$i \rightarrow i$</td>
<td>eliminator</td>
</tr>
<tr>
<td>T</td>
<td>$t$</td>
<td>singleton type</td>
</tr>
<tr>
<td>tt</td>
<td>$i$</td>
<td>constructor</td>
</tr>
</tbody>
</table>

Figure 5.1: The constants of CTT
CHAPTER 5. CONSTRUCTIVE TYPE THEORY

CTT supports all of Type Theory apart from list types, well-ordering types, and universes. Universes could be introduced à la Tarski, adding new constants as names for types. The formulation à la Russell, where types denote themselves, is only possible if we identify the meta-types \( i \) and \( t \). Most published formulations of well-ordering types have difficulties involving extensionality of functions; I suggest that you use some other method for defining recursive types. List types are easy to introduce by declaring new rules.

CTT uses the 1982 version of Type Theory, with extensional equality. The computation \( a = b \in A \) and the equality \( c \in Eq(A, a, b) \) are interchangeable. Its rewriting tactics prove theorems of the form \( a = b \in A \). It could be modified to have intensional equality, but rewriting tactics would have to prove theorems of the form \( c \in Eq(A, a, b) \) and the computation rules might require a separate simplifier.

5.1 Syntax

The constants are shown in Fig. 5.1. The infixes include the function application operator (sometimes called ‘apply’), and the 2-place type operators. Note that meta-level abstraction and application, \( \lambda x. b \) and \( f(a) \), differ from object-level abstraction and application, \( \mathbf{lam} \ x. \ b \) and \( b \ a \). A CTT function \( f \) is simply an individual as far as Isabelle is concerned: its Isabelle type is \( i \), not say \( i \Rightarrow i \).

The notation for CTT (Fig. 5.2) is based on that of Nordström et al. [11]. The empty type is called \( F \) and the one-element type is \( T \); other finite types are built as \( T + T + T \), etc.

Quantification is expressed by sums \( \sum_{x \in A} B[x] \) and products \( \prod_{x \in A} B[x] \). Instead of \( \mathbf{Sum}(A, B) \) and \( \mathbf{Prod}(A, B) \) we may write \( \mathbf{SUM} \ x:A. \ B[x] \) and \( \mathbf{PROD} \ x:A. \ B[x] \). For example, we may write

\[
\mathbf{SUM} \ y:B. \ \mathbf{PROD} \ x:A. \ C(x,y) \quad \text{for} \quad \mathbf{Sum}(B, \%y. \ \mathbf{Prod}(A, \%x. \ C(x,y)))
\]

The special cases as \( A \times B \) and \( A \rightarrow B \) abbreviate general sums and products over a constant family.\(^1\) Isabelle accepts these abbreviations in parsing and uses them whenever possible for printing.

\(^1\)Unlike normal infix operators, \( \times \) and \( \rightarrow \) merely define abbreviations; there are no constants \( \mathbf{op} \times \) and \( \mathbf{op} \rightarrow \).
### Chapter 5. Constructive Type Theory

<table>
<thead>
<tr>
<th>symbol</th>
<th>name</th>
<th>meta-type</th>
<th>priority</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lam</td>
<td>lambda</td>
<td>((i \Rightarrow o) \Rightarrow i)</td>
<td>10</td>
<td>(\lambda)-abstraction</td>
</tr>
</tbody>
</table>

#### Binders

<table>
<thead>
<tr>
<th>symbol</th>
<th>meta-type</th>
<th>priority</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(_)</td>
<td>([i, i] \rightarrow i)</td>
<td>Left 55</td>
<td>function application</td>
</tr>
<tr>
<td>+</td>
<td>([t, t] \rightarrow t)</td>
<td>Right 30</td>
<td>sum of two types</td>
</tr>
</tbody>
</table>

#### Infixes

<table>
<thead>
<tr>
<th>external</th>
<th>internal</th>
<th>standard notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROD</td>
<td>(\prod x:A . B[x])</td>
<td>product (\Pi_{x:A} B[x])</td>
</tr>
<tr>
<td>SUM</td>
<td>(\sum x:A . B[x])</td>
<td>sum (\Sigma_{x:A} B[x])</td>
</tr>
<tr>
<td>A (\rightarrow) B</td>
<td>Prod(A, (\lambda x.B[x]))</td>
<td>function space (A \rightarrow B)</td>
</tr>
<tr>
<td>A (\ast) B</td>
<td>Sum(A, (\lambda x.B[x]))</td>
<td>binary product (A \times B)</td>
</tr>
</tbody>
</table>

#### Translations

- \(\text{prop} = \text{type type}\)
- \(\text{type} = \text{expression of type } t\)
- \(\text{term} = \text{expression of type } i\)

#### Grammar

Figure 5.2: Syntax of CTT
refl_type \quad \text{A type} \implies \text{A = A}
refl_elem \quad \text{a : A} \implies \text{a = a : A}

sym_type \quad \text{A = B} \implies \text{B = A}
sym_elem \quad \text{a = b : A} \implies \text{b = a : A}

trans_type \quad [\mid \text{A = B; B = C} \mid] \implies \text{A = C}
trans_elem \quad [\mid \text{a = b : A; b = c : A} \mid] \implies \text{a = c : A}

equal_types \quad [\mid \text{a : A; A = B} \mid] \implies \text{a : B}
equal_typesL \quad [\mid \text{a = b : A; A = B} \mid] \implies \text{a = b : B}

subst_type \quad [\mid \text{a : A; !!z.z:A} \implies \text{B(z)} \text{ type} \mid] \implies \text{B(a) type}
subst_typeL \quad [\mid \text{a = c : A; !!z.z:A} \implies \text{B(z)} = \text{D(z)} \mid] \implies \text{B(a)} = \text{D(c)}

subst_elem \quad [\mid \text{a : A; !!z.z:A} \implies \text{b(z):B(z) }\mid] \implies \text{b(a):B(a)}
subst_elemL \quad [\mid \text{a = c : A; !!z.z:A} \implies \text{b(z) = d(z) : B(z)} \mid] \implies \text{b(a) = d(c) : B(a)}

refl_red \quad \text{Reduce(a,a)}
red_if_equal \quad \text{a = b : A} \implies \text{Reduce(a,b)}
trans_red \quad [\mid \text{a = b : A; Reduce(b,c) }\mid] \implies \text{a = c : A}

Figure 5.3: General equality rules
CHAPTER 5. CONSTRUCTIVE TYPE THEORY

NF  N type

NIO  0 : N

NI_succ  a : N => succ(a) : N

NI_succL  a = b : N => succ(a) = succ(b) : N

NE  [| p : N; a : C(0);
   !!u v. [| u: N; v: C(u) |] => b(u,v): C(succ(u))
   |] => rec(p, a, %u v. b(u,v)) : C(p)

NEL  [| p = q : N; a = c : C(0);
   !!u v. [| u: N; v: C(u) |] => b(u,v) = d(u,v): C(succ(u))
   |] => rec(p, a, %u v. b(u,v)) = rec(q,c,d) : C(p)

NCO  [| a : C(0);
   !!u v. [| u: N; v: C(u) |] => b(u,v): C(succ(u))
   |] => rec(0, a, %u v. b(u,v)) = a : C(0)

NC_succ  [| p : N; a : C(0);
   !!u v. [| u: N; v: C(u) |] => b(u,v): C(succ(u))
   |] => rec(succ(p), a, %u v. b(u,v)) = b(p, rec(p, a, %u v. b(u,v))) : C(succ(p))

zero_ne_succ  [| a : N; 0 = succ(a) : N |] => 0 : F

Figure 5.4: Rules for type \( N \)

ProdF  [| A type; !!x. x:A => B(x) type |] => PROD x:A. B(x) type

ProdFL  [| A = C; !!x. x:A => B(x) = D(x) |] =>
   PROD x:A. B(x) = PROD x:C. D(x)

ProdI  [| A type; !!x. x:A => b(x):B(x)
   |] => lam x. b(x) : PROD x:A. B(x)

ProdIL  [| A type; !!x. x:A => b(x) = c(x) : B(x)
   |] => lam x. b(x) = lam x. c(x) : PROD x:A. B(x)

ProdE  [| p : PROD x:A. B(x); a : A |] => p'a : B(a)

ProdEL  [| p=q: PROD x:A. B(x); a=b : A |] => p'a = q'b : B(a)

ProdC  [| a : A; !!x. x:A => b(x) : B(x)
   |] => (lam x. b(x)) ' a = b(a) : B(a)

ProdC2  p : PROD x:A. B(x) => (lam x. p'x) = p : PROD x:A. B(x)

Figure 5.5: Rules for the product type \( \prod_{x \in A} B[x] \)
\[
\text{SumF} \quad [\text{A type}; \quad \forall x. \ x : A \Rightarrow B(x) \text{ type}] \Rightarrow \text{SUM } x : A. \ B(x) \text{ type}
\]

\[
\text{SumFL} \quad [\text{A = C}; \quad \forall x. \ x : A \Rightarrow B(x) = D(x)] \Rightarrow \text{SUM } x : A. \ B(x) = \text{SUM } x : C. \ D(x)
\]

\[
\text{SumI} \quad [\text{a : A}; \quad \text{b : B(a)}] \Rightarrow \langle a, b \rangle : \text{SUM } x : A. \ B(x)
\]

\[
\text{SumIL} \quad [\text{a=c:A}; \quad \text{b=d:B(a)}] \Rightarrow \langle a, b \rangle = \langle c, d \rangle : \text{SUM } x : A. \ B(x)
\]

\[
\text{SumE} \quad [\text{p: SUM x:A. B(x)}; \quad \forall x \ y. \ [\text{x:A}; \ y:B(x)] \Rightarrow c(x,y) : C(<x,y>)] \Rightarrow \text{split}(p, \ %x \ y. \ c(x,y)) : C(p)
\]

\[
\text{SumEL} \quad [\text{p=q : SUM x:A. B(x)}; \quad \forall x \ y. \ [\text{x:A}; \ y:B(x)] \Rightarrow c(x,y)=d(x,y) : C(<x,y>)] \Rightarrow \text{split}(p, \ %x \ y. \ c(x,y)) = \text{split}(q, \ %x \ y. \ d(x,y)) : C(p)
\]

\[
\text{SumC} \quad [\text{a : A}; \quad \text{b : B(a)}; \quad \forall x \ y. \ [\text{x:A}; \ y:B(x)] \Rightarrow c(x,y) : C(<x,y>)] \Rightarrow \text{split}(\langle a,b \rangle, \ %x \ y. \ c(x,y)) = c(a,b) : C(\langle a,b \rangle)
\]

\[
\text{fst_def} \quad \text{fst}(a) == \text{split}(a, \ %x \ y. \ x)
\]

\[
\text{snd_def} \quad \text{snd}(a) == \text{split}(a, \ %x \ y. \ y)
\]

Figure 5.6: Rules for the sum type \( \sum_{x \in A} B[x] \)

### 5.2 Rules of inference

The rules obey the following naming conventions. Type formation rules have the suffix \(F\). Introduction rules have the suffix \(I\). Elimination rules have the suffix \(E\). Computation rules, which describe the reduction of eliminators, have the suffix \(C\). The equality versions of the rules (which permit reductions on subterms) are called long rules; their names have the suffix \(L\). Introduction and computation rules are often further suffixed with constructor names.

Figure 5.3 presents the equality rules. Most of them are straightforward: reflexivity, symmetry, transitivity and substitution. The judgement \(\text{Reduce}\) does not belong to Type Theory proper; it has been added to implement rewriting. The judgement \(\text{Reduce}(a, b)\) holds when \(a = b : A\) holds. It also holds when \(a\) and \(b\) are syntactically identical, even if they are ill-typed, because rule \(\text{refl_red}\) does not verify that \(a\) belongs to \(A\).

The \(\text{Reduce}\) rules do not give rise to new theorems about the standard judgements. The only rule with \(\text{Reduce}\) in a premise is \(\text{trans_red}\), whose other premise ensures that \(a\) and \(b\) (and thus \(c\)) are well-typed.

Figure 5.4 presents the rules for \(N\), the type of natural numbers. They include \(\text{zero_ne_succ}\), which asserts \(0 \neq n + 1\). This is the fourth Peano
\textbf{CHAPTER 5. CONSTRUCTIVE TYPE THEORY} 70

\begin{align*}
\text{PlusF} & \quad [\mid A \text{ type}; B \text{ type} \mid] \implies A+B \text{ type} \\
\text{PlusFL} & \quad [\mid A = C; B = D \mid] \implies A+B = C+D \\
\text{PlusI}_{\text{inl}} & \quad [\mid a : A; B \text{ type} \mid] \implies \text{inl}(a) : A+B \\
\text{PlusI}_{\text{inlL}} & \quad [\mid a = c : A; B \text{ type} \mid] \implies \text{inl}(a) = \text{inl}(c) : A+B \\
\text{PlusI}_{\text{inr}} & \quad [\mid A \text{ type}; b : B \mid] \implies \text{inr}(b) : A+B \\
\text{PlusI}_{\text{inrL}} & \quad [\mid A \text{ type}; b = d : B \mid] \implies \text{inr}(b) = \text{inr}(d) : A+B \\
\text{PlusE} & \quad [\mid p: A+B; \quad \forall x. x:A \implies c(x): C(inl(x)); \quad \forall y. y:B \implies d(y): C(inr(y)) \mid] \implies \text{when}(p, \forall x. c(x), \forall y. d(y)) : C(p) \\
\text{PlusEL} & \quad [\mid p = q : A+B; \quad \forall x. x:A \implies c(x) = e(x) : C(inl(x)); \quad \forall y. y:B \implies d(y) = f(y) : C(inr(y)) \mid] \implies \text{when}(p, \forall x. c(x), \forall y. d(y)) = \text{when}(q, \forall x. e(x), \forall y. f(y)) : C(p) \\
\text{PlusC}_{\text{inl}} & \quad [\mid a : A; \quad \forall x. x:A \implies c(x): C(inl(x)); \quad \forall y. y:B \implies d(y): C(inr(y)) \mid] \implies \text{when}(\text{inl}(a), \forall x. c(x), \forall y. d(y)) = c(a) : C(inl(a)) \\
\text{PlusC}_{\text{inr}} & \quad [\mid b : B; \quad \forall x. x:A \implies c(x): C(inl(x)); \quad \forall y. y:B \implies d(y): C(inr(y)) \mid] \implies \text{when}(\text{inr}(b), \forall x. c(x), \forall y. d(y)) = d(b) : C(inr(b))
\end{align*}

Figure 5.7: Rules for the binary sum type $A + B$

\begin{align*}
\text{FF} & \quad \text{F type} \\
\text{FE} & \quad [\mid p: F; C \text{ type} \mid] \implies \text{contr}(p) : C \\
\text{FEL} & \quad [\mid p = q : F; C \text{ type} \mid] \implies \text{contr}(p) = \text{contr}(q) : C \\
\text{TF} & \quad \text{T type} \\
\text{TI} & \quad \text{tt} : T \\
\text{TE} & \quad [\mid p : T; c : C(\text{tt}) \mid] \implies c : C(p) \\
\text{TEL} & \quad [\mid p = q : T; c = d : C(\text{tt}) \mid] \implies c = d : C(p) \\
\text{TC} & \quad p : T \implies p = \text{tt} : T
\end{align*}

Figure 5.8: Rules for types $F$ and $T$
CHAPTER 5. CONSTRUCTIVE TYPE THEORY

EqF \[ | A \text{ type}; \ a : A; \ b : A | \implies Eq(A,a,b) \text{ type} \]
EqFL \[ | A=B; \ a=c: A; \ b=d : A | \implies Eq(A,a,b) = Eq(B,c,d) \]
EqI \[ a = b : A \implies eq : Eq(A,a,b) \]
EqE \[ p : Eq(A,a,b) \implies a = b : A \]
EqC \[ p : Eq(A,a,b) \implies p = eq : Eq(A,a,b) \]

Figure 5.9: Rules for the equality type \( Eq(A,a,b) \)

replace_type \[ | B = A; \ a : A | \implies a : B \]
subst_eqtyparg \[ | a=c : A; \ !!z. z:A \implies B(z) \text{ type } | \implies B(a)=B(c) \]

Figure 5.10: Derived rules for CTT

The constant \textbf{rec} constructs proof terms when mathematical induction, rule \textbf{NE}, is applied. It can also express primitive recursion. Since \textbf{rec} can be applied to higher-order functions, it can even express Ackermann’s function, which is not primitive recursive [18, page 104].

Figure 5.5 shows the rules for general product types, which include function types as a special case. The rules correspond to the predicate calculus rules for universal quantifiers and implication. They also permit reasoning about functions, with the rules of a typed \( \lambda \)-calculus.

Figure 5.6 shows the rules for general sum types, which include binary product types as a special case. The rules correspond to the predicate calculus rules for existential quantifiers and conjunction. They also permit reasoning about ordered pairs, with the projections \textbf{fst} and \textbf{snd}.

Figure 5.7 shows the rules for binary sum types. They correspond to the predicate calculus rules for disjunction. They also permit reasoning about disjoint sums, with the injections \textbf{inl} and \textbf{inr} and case analysis operator \textbf{when}.

Figure 5.8 shows the rules for the empty and unit types, \( F \) and \( T \). They correspond to the predicate calculus rules for absurdity and truth.
Figure 5.9 shows the rules for equality types. If \( a = b \in A \) is provable then \( \text{eq} \) is a canonical element of the type \( \text{Eq}(A, a, b) \), and vice versa. These rules define extensional equality; the most recent versions of Type Theory use intensional equality [11].

Figure 5.10 presents the derived rules. The rule \( \text{subst\_prodE} \) is derived from \( \text{prodE} \), and is easier to use in backwards proof. The rules \( \text{SumE\_fst} \) and \( \text{SumE\_snd} \) express the typing of \( \text{fst} \) and \( \text{snd} \); together, they are roughly equivalent to \( \text{SumE} \) with the advantage of creating no parameters. Section 5.12 below demonstrates these rules in a proof of the Axiom of Choice.

All the rules are given in \( \eta \)-expanded form. For instance, every occurrence of \( \lambda u.v. \ b(u, v) \) could be abbreviated to \( b \) in the rules for \( N \). The expanded form permits Isabelle to preserve bound variable names during backward proof. Names of bound variables in the conclusion (here, \( u \) and \( v \)) are matched with corresponding bound variables in the premises.

### 5.3 Rule lists

The Type Theory tactics provide rewriting, type inference, and logical reasoning. Many proof procedures work by repeatedly resolving certain Type Theory rules against a proof state. CTT defines lists — each with type \( \text{thm list} \) — of related rules.

- \( \text{form\_rls} \) contains formation rules for the types \( N, \Pi, \Sigma, +, \text{Eq}, \text{F}, \) and \( T \).
- \( \text{formL\_rls} \) contains long formation rules for \( \Pi, \Sigma, +, \) and \( \text{Eq} \). (For other types use \( \text{refl\_type} \).)
- \( \text{intr\_rls} \) contains introduction rules for the types \( N, \Pi, \Sigma, +, \) and \( T \).
- \( \text{intrL\_rls} \) contains long introduction rules for \( N, \Pi, \Sigma, \) and \( + \). (For \( T \) use \( \text{refl\_elem} \).)
- \( \text{elim\_rls} \) contains elimination rules for the types \( N, \Pi, \Sigma, +, \) and \( F \). The rules for \( \text{Eq} \) and \( T \) are omitted because they involve no eliminator.
- \( \text{elimL\_rls} \) contains long elimination rules for \( N, \Pi, \Sigma, +, \) and \( F \).
- \( \text{comp\_rls} \) contains computation rules for the types \( N, \Pi, \Sigma, \) and \( + \). Those for \( \text{Eq} \) and \( T \) involve no eliminator.
- \( \text{basic\_defs} \) contains the definitions of \( \text{fst} \) and \( \text{snd} \).
5.4 Tactics for subgoal reordering

\begin{verbatim}
test_assume_tac : int -> tactic
typechk_tac : thm list -> tactic
equal_tac : thm list -> tactic
intr_tac : thm list -> tactic
\end{verbatim}

Blind application of CTT rules seldom leads to a proof. The elimination rules, especially, create subgoals containing new unknowns. These subgoals unify with anything, creating a huge search space. The standard tactic \texttt{filt\_resolve\_tac} (see the \textit{Reference Manual}) fails for goals that are too flexible; so does the CTT tactic \texttt{test\_assume\_tac}. Used with the tactical \texttt{REPEAT\_FIRST} they achieve a simple kind of subgoal reordering: the less flexible subgoals are attempted first. Do some single step proofs, or study the examples below, to see why this is necessary.

\texttt{test\_assume\_tac} \textit{i} uses \texttt{assume\_tac} to solve the subgoal by assumption, but only if subgoal \textit{i} has the form \(a \in A\) and the head of \(a\) is not an unknown. Otherwise, it fails.

\texttt{typechk\_tac} \textit{thms} uses \texttt{thms} with formation, introduction, and elimination rules to check the typing of constructions. It is designed to solve goals of the form \(a \in ?A\), where \(a\) is rigid and \(?A\) is flexible; thus it performs type inference. The tactic can also solve goals of the form \(A\) type.

\texttt{equal\_tac} \textit{thms} uses \texttt{thms} with the long introduction and elimination rules to solve goals of the form \(a = b \in A\), where \(a\) is rigid. It is intended for deriving the long rules for defined constants such as the arithmetic operators. The tactic can also perform type-checking.

\texttt{intr\_tac} \textit{thms} uses \texttt{thms} with the introduction rules to break down a type. It is designed for goals like \(?a \in A\) where \(?a\) is flexible and \(A\) rigid. These typically arise when trying to prove a proposition \(A\), expressed as a type.

5.5 Rewriting tactics

\begin{verbatim}
rew_tac : thm list -> tactic
hyp\_rew\_tac : thm list -> tactic
\end{verbatim}

Object-level simplification is accomplished through proof, using the CTT equality rules and the built-in rewriting functor \texttt{T\textsc{SimpFun}}.\footnote{This should not be confused with Isabelle’s main simplifier; \texttt{T\textsc{SimpFun}} is only useful for CTT and similar logics with type inference rules. At present it is undocumented.} The rewrites
include the computation rules and other equations. The long versions of the other rules permit rewriting of subterms and subtypes. Also used are transitivity and the extra judgement form Reduce. Meta-level simplification handles only definitional equality.

rew_tac thms applies thms and the computation rules as left-to-right rewrites. It solves the goal \( a = b \in A \) by rewriting \( a \) to \( b \). If \( b \) is an unknown then it is assigned the rewritten form of \( a \). All subgoals are rewritten.

hyp_rew_tac thms is like rew_tac, but includes as rewrites any equations present in the assumptions.

5.6 Tactics for logical reasoning

Interpreting propositions as types lets CTT express statements of intuitionistic logic. However, Constructive Type Theory is not just another syntax for first-order logic. There are fundamental differences.

Can assumptions be deleted after use? Not every occurrence of a type represents a proposition, and Type Theory assumptions declare variables. In first-order logic, \( \lor \)-elimination with the assumption \( P \lor Q \) creates one subgoal assuming \( P \) and another assuming \( Q \), and \( P \lor Q \) can be deleted safely. In Type Theory, \( + \)-elimination with the assumption \( z \in A + B \) creates one subgoal assuming \( x \in A \) and another assuming \( y \in B \) (for arbitrary \( x \) and \( y \)). Deleting \( z \in A + B \) when other assumptions refer to \( z \) may render the subgoal unprovable: arguably, meaningless.

Isabelle provides several tactics for predicate calculus reasoning in CTT:

```
mp_tac  : int -> tactic
add_mp_tac  : int -> tactic
safe_step_tac : thm list -> int -> tactic
safe_tac : thm list -> int -> tactic
step_tac : thm list -> int -> tactic
pc_tac : thm list -> int -> tactic
```

These are loosely based on the intuitionistic proof procedures of FOL. For the reasons discussed above, a rule that is safe for propositional reasoning may be unsafe for type-checking; thus, some of the ‘safe’ tactics are misnamed.

mp_tac \( i \) searches in subgoal \( i \) for assumptions of the form \( f \in \Pi(A,B) \) and \( a \in A \), where \( A \) may be found by unification. It replaces \( f \in \Pi(A,B) \) by \( z \in B(a) \), where \( z \) is a new parameter. The tactic can produce multiple outcomes for each suitable pair of assumptions. In short, mp_tac performs Modus Ponens among the assumptions.
add_mp_tac \( i \) is like mp_tac \( i \) but retains the assumption \( f \in \Pi(A,B) \). It avoids information loss but obviously loops if repeated.

safestep_tac \( thms \) \( i \) attacks subgoal \( i \) using formation rules and certain other ‘safe’ rules (FE, ProdI, SumE, PlusE), calling mp_tac when appropriate. It also uses \( thms \), which are typically premises of the rule being derived.

safe_tac \( thms \) \( i \) attempts to solve subgoal \( i \) by means of backtracking, using safestep_tac.

step_tac \( thms \) \( i \) tries to reduce subgoal \( i \) using safestep_tac, then tries unsafe rules. It may produce multiple outcomes.

pc_tac \( thms \) \( i \) tries to solve subgoal \( i \) by backtracking, using step_tac.

### 5.7 A theory of arithmetic

**Arith** is a theory of elementary arithmetic. It proves the properties of addition, multiplication, subtraction, division, and remainder, culminating in the theorem

\[ a \mod b + (a/b) \times b = a. \]

Figure 5.11 presents the definitions and some of the key theorems, including commutative, distributive, and associative laws.

The operators #+, -, |-|, #*, mod and div stand for sum, difference, absolute difference, product, remainder and quotient, respectively. Since Type Theory has only primitive recursion, some of their definitions may be obscure.

The difference \( a - b \) is computed by taking \( b \) predecessors of \( a \), where the predecessor function is \( \lambda v . \text{rec}(v, 0, \lambda x y . x) \).

The remainder \( a \mod b \) counts up to \( a \) in a cyclic fashion, using \( 0 \) as the successor of \( b-1 \). Absolute difference is used to test the equality \( \text{succ}(v) = b \).

The quotient \( a/b \) is computed by adding one for every number \( x \) such that \( 0 \leq x \leq a \) and \( x \mod b = 0 \).

### 5.8 The examples directory

This directory contains examples and experimental proofs in CTT.

CTT/ex/typechk.ML contains simple examples of type-checking and type deduction.
### Table 5.11: The theory of arithmetic

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meta-Type</th>
<th>Priority</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#*</code></td>
<td><code>[i, i]</code></td>
<td>70</td>
<td>Left Multiplication</td>
</tr>
<tr>
<td><code>div</code></td>
<td><code>[i, i]</code></td>
<td>70</td>
<td>Left Division</td>
</tr>
<tr>
<td><code>mod</code></td>
<td><code>[i, i]</code></td>
<td>70</td>
<td>Left Modulus</td>
</tr>
<tr>
<td><code>#+</code></td>
<td><code>[i, i]</code></td>
<td>65</td>
<td>Left Addition</td>
</tr>
<tr>
<td><code>-</code></td>
<td><code>[i, i]</code></td>
<td>65</td>
<td>Left Subtraction</td>
</tr>
<tr>
<td>`</td>
<td>`</td>
<td><code>[i, i]</code></td>
<td>65</td>
</tr>
</tbody>
</table>

**Addition Definitions**
- \( a#+b = \text{rec}(a, b, \%u v. \text{succ}(v)) \)
- \( a-b = \text{rec}(b, a, \%u v. \text{rec}(v, 0, \%x y. x)) \)
- \( a|-|b = (a-b) #+ (b-a) \)
- \( a#*b = \text{rec}(a, 0, \%u v. b #+ v) \)

**Division Definitions**
- \( a \text{ div } b = \text{rec}(a, 0, \%u v. \text{rec}(\text{succ}(v) |-| b, 0, \%x y. \text{succ}(v))) \)

**Addition Typing**
- \([| a:N; b:N |] \Rightarrow a #+ b : N \)
- \( a #+ b = b #+ a : N \)

**Addition Properties**
- \( \text{add_assoc} ([| a:N; b:N; c:N |]) \Rightarrow (a #+ b) #+ c = a #+ (b #+ c) : N \)
- \( \text{add_commute} ([| a:N; b:N |]) \Rightarrow a #+ b = b #+ a : N \)

**Multiplication Definitions**
- \( a#*b = \text{rec}(a, 0, \%u v. \text{succ}(v)) \)
- \( a#*b = b #+ (a#*b) : N \)

**Multiplication Typing**
- \([| a:N; b:N |] \Rightarrow a #* b : N \)
- \( a #* b = b #* a : N \)

**Multiplication Properties**
- \( \text{mult_assoc} ([| a:N; b:N; c:N |]) \Rightarrow (a #* b) #* c = a #* (b #* c) : N \)
- \( \text{mult_commute} ([| a:N; b:N |]) \Rightarrow a #* b = b #* a : N \)

**Subtraction Definitions**
- \( a |-| b = (a-b) #+ (b-a) \)

**Subtraction Typing**
- \([| a:N; b:N |] \Rightarrow a - b : N \)

**Subtraction Properties**
- \( \text{add_inverse_diff} ([| a:N; b:N; b-a=0 : N |] \Rightarrow b #+ (a-b) = a : N \)

Figure 5.11: The theory of arithmetic
CTT/ex/elim.ML contains some examples from Martin-Löf [9], proved using 
\texttt{pc_tac}.

CTT/ex/equal.ML contains simple examples of rewriting.

CTT/ex/synth.ML demonstrates the use of unknowns with some trivial ex-
amples of program synthesis.

5.9 Example: type inference

Type inference involves proving a goal of the form $a \in \tau A$, where $a$ is a term 
and $\tau A$ is an unknown standing for its type. The type, initially unknown, 
takes shape in the course of the proof. Our example is the predecessor func-
tion on the natural numbers.

\begin{verbatim}
Goal "lam n. rec(n, 0, %x y. x) : ?A";
Level 0
  lam n. rec(n,0,%x y. x) : ?A
1. lam n. rec(n,0,%x y. x) : ?A

Since the term is a Constructive Type Theory $\lambda$-abstraction (not to be 
confused with a meta-level abstraction), we apply the rule \texttt{ProdI}, for $\Pi$-
introduction. This instantiates $\tau A$ to a product type of unknown domain 
and range.

by (resolve_tac [ProdI] 1);
Level 1
  lam n. rec(n,0,%x y. x) : PROD x:?A1. ?B1(x)
1. ?A1 type
2. !!n. n : ?A1 ==> rec(n,0,%x y. x) : ?B1(n)

Subgoal 1 is too flexible. It can be solved by instantiating $\tau A_1$ to any type, 
but most instantiations will invalidate subgoal 2. We therefore tackle the 
latter subgoal. It asks the type of a term beginning with \texttt{rec}, which can be 
found by $N$-elimination.

by (eresolve_tac [NE] 2);
Level 2
  lam n. rec(n,0,%x y. x) : PROD x:N. ?C2(x,x)
1. N type
2. !!n. 0 : ?C2(n,0)
3. !!n x y. [| x : N; y : ?C2(n,x) |] ==> x : ?C2(n,succ(x))

Subgoal 1 is no longer flexible: we now know $\tau A_1$ is the type of natural 
numbers. However, let us continue proving nontrivial subgoals. Subgoal 2 
asks, what is the type of 0?
by (resolve_tac [N10] 2);

Level 3
lam n. rec(n,0,\%x y. x) : N --> N
1. N type
2. \forall n x y. [| x : N; y : N |] ==\> x : N

The type ?A is now fully determined. It is the product type \( \prod_{x \in N} N \), which is shown as the function type \( N \to N \) because there is no dependence on \( x \). But we must prove all the subgoals to show that the original term is validly typed. Subgoal 2 is provable by assumption and the remaining subgoal falls by \( N \)-formation.

by (assume_tac 2);

Level 4
lam n. rec(n,0,\%x y. x) : N --> N
1. N type
by (resolve_tac [NF] 1);

Level 5
lam n. rec(n,0,\%x y. x) : N --> N
No subgoals!

Calling \texttt{typechk_tac} can prove this theorem in one step.

Even if the original term is ill-typed, one can infer a type for it, but unprovable subgoals will be left. As an exercise, try to prove the following invalid goal:

\texttt{Goal "lam n. rec(n, 0, \%x y. tt) : ?A";}

5.10 An example of logical reasoning

Logical reasoning in Type Theory involves proving a goal of the form \( ?a \in A \), where type \( A \) expresses a proposition and \( ?a \) stands for its proof term, a value of type \( A \). The proof term is initially unknown and takes shape during the proof.

Our example expresses a theorem about quantifiers in a sorted logic:

\[
\exists x \in A . P(x) \lor Q(x) \\
(\exists x \in A . P(x)) \lor (\exists x \in A . Q(x))
\]

By the propositions-as-types principle, this is encoded using \( \Sigma \) and \(+\) types. A special case of it expresses a distributive law of Type Theory:

\[
A \times (B + C) \\
(A \times B) + (A \times C)
\]
Generalizing this from $\times$ to $\Sigma$, and making the typing conditions explicit, yields the rule we must derive:

$$
\begin{array}{c}
\ldots \\
A \text{ type} \quad B(x) \text{ type} \quad C(x) \text{ type} \\
\ldots
\end{array}
\quad
\begin{array}{c}
[x \in A] \\
\vdash ?a \in (\sum_{x \in A} B(x)) + (\sum_{x \in A} C(x))
\end{array}
$$

To begin, we bind the rule’s premises — returned by the goal command — to the ML variable prems.

```ml
val prems = Goal "[| A type; \\
!!x. x:A ==> B(x) type; \\
!!x. x:A ==> C(x) type; \\
p: \sum x:A. B(x) + C(x) |
|] ==> ?a : (\sum x:A. B(x)) + (\sum x:A. C(x))";
```

The last premise involves the sum type $\Sigma$. Since it is a premise rather than the assumption of a goal, it cannot be found by eresolve_tac. We could insert it (and the other atomic premise) by calling

```ml
cut_facts_tac prems 1;
```

A forward proof step is more straightforward here. Let us resolve the $\Sigma$-elimination rule with the premises using RL. This inference yields one result, which we supply to resolve_tac.

```ml
by (resolve_tac (prems RL [SumE]) 1);
```

The subgoal has two new parameters, $x$ and $y$. In the main goal, $?a$ has been instantiated with a split term. The assumption $y \in B(x) + C(x)$ is eliminated next, causing a case split and creating the parameter $xa$. This inference also inserts when into the main goal.
by (eresolve_tac [PlusE] 1);  
Level 2  
\[ \text{split}(p,\%x\ y. \ when(y,?c2(x,y),?d2(x,y))) \]  
: (SUM x:A. \( B(x) \)) + (SUM x:A. \( C(x) \))  
1. !!x y xa.  
\[ [| x : A; xa : B(x) |] \implies \]  
\( ?c2(x,y,xa) : \text{SUM} x:A. \( B(x) \) \) + (SUM x:A. \( C(x) \))  
2. !!x y ya.  
\[ [| x : A; ya : C(x) |] \implies \]  
\( ?d2(x,y,ya) : \text{SUM} x:A. \( B(x) \) \) + (SUM x:A. \( C(x) \))  

To complete the proof object for the main goal, we need to instantiate the terms \( ?c2(x, y, xa) \) and \( ?d2(x, y, xa) \). We attack subgoal 1 by a \(+\)-introduction rule; since the goal assumes \( xa \in B(x) \), we take the left injection (\( \text{inl} \)).

by (resolve_tac [PlusI_inl] 1);  
Level 3  
\[ \text{split}(p,\%x\ y. \ when(y,\%xa. \text{inl}(?a3(x,y,xa)),?d2(x,y))) \]  
: (SUM x:A. \( B(x) \)) + (SUM x:A. \( C(x) \))  
1. !!x y xa.  
\[ [| x : A; xa : B(x) |] \implies \]  
\( ?a3(x,y,xa) : \text{SUM} x:A. \( B(x) \) \)  
2. !!x y xa.  
\[ [| x : A; xa : B(x) |] \implies \]  
\( \text{SUM} x:A. \( C(x) \) \) type  
3. !!x y ya.  
\[ [| x : A; ya : C(x) |] \implies \]  
\( ?d2(x,y,ya) : \text{SUM} x:A. \( B(x) \) \) + (SUM x:A. \( C(x) \))  

A new subgoal 2 has appeared, to verify that \( \Sigma_{x \in A} C(x) \) is a type. Continuing to work on subgoal 1, we apply the \( \Sigma \)-introduction rule. This instantiates the term \( ?a3(x, y, xa) \); the main goal now contains an ordered pair, whose components are two new unknowns.

by (resolve_tac [SumI] 1);  
Level 4  
\[ \text{split}(p,\%x\ y. \ when(y,\%xa. \text{inl}(<?a4(x,y,xa),?b4(x,y,xa)>)?,?d2(x,y))) \]  
: (SUM x:A. \( B(x) \)) + (SUM x:A. \( C(x) \))  
1. !!x y xa.  
\[ [| x : A; xa : B(x) |] \implies \]  
\( ?a4(x,y,xa) : A \)  
2. !!x y xa.  
\[ [| x : A; xa : B(x) |] \implies \]  
\( ?b4(x,y,xa) : B(?a4(x,y,xa)) \)  
3. !!x y xa.  
\[ [| x : A; xa : B(x) |] \implies \]  
\( \text{SUM} x:A. \( C(x) \) \) type  
4. !!x y ya.  
\[ [| x : A; ya : C(x) |] \implies \]  
\( ?d2(x,y,ya) : (\text{SUM} x:A. \( B(x) \)) + (SUM x:A. \( C(x) \)) \)  

The two new subgoals both hold by assumption. Observe how the unknowns \( ?a4 \) and \( ?b4 \) are instantiated throughout the proof state.

by (assume_tac 1);  
Level 5  
\[ \text{split}(p,\%x\ y. \ when(y,\%xa. \text{inl}(<?x,?b4(x,y,xa)>)?),?d2(x,y))) \]  
: (SUM x:A. \( B(x) \)) + (SUM x:A. \( C(x) \))
1. \( \forall x \, y \, x. \, [\| x : A; \, x a : B(x) \|] \Rightarrow ?b4(x,y,xa) : B(x) \)
2. \( \forall x \, y \, x a. \, [\| x : A; \, x a : B(x) \|] \Rightarrow \sum x:A. \, C(x) \text{ type} \)
3. \( \forall x \, y \, y a. \, [\| x : A; \, y a : C(x) \|] \Rightarrow \sum x:A. \, B(x) + \sum x:A. \, C(x) \)

Subgoal 1 is an example of a well-formedness subgoal [3]. Such subgoals are usually trivial; this one yields to `typechk_tac`, given the current list of premises.

by (typechk_tac prems);
Level 7
split(p,%x y. when(y,%xa. inl(<x,xa>),?d2(x,y)))
1. \( \forall x \, y \, x a. \, [\| x : A; \, x a : B(x) \|] \Rightarrow \sum x:A. \, C(x) \text{ type} \)
2. \( \forall x \, y \, y a. \, [\| x : A; \, y a : C(x) \|] \Rightarrow \sum x:A. \, B(x) + \sum x:A. \, C(x) \)

This subgoal is the other case from the +-elimination above, and can be proved similarly. Quicker is to apply `pc_tac`. The main goal finally gets a fully instantiated proof object.

by (pc_tac prems 1);
Level 8
split(p,%x y. when(y,%xa. inl(<x,xa>),%y. inr(<x,y>)))
: (SUM x:A. \, B(x)) + (SUM x:A. \, C(x))

No subgoals!

Calling `pc_tac` after the first \(\Sigma\)-elimination above also proves this theorem.

### 5.11 Example: deriving a currying functional

In simply-typed languages such as ML, a currying functional has the type

\[ (A \times B \rightarrow C) \rightarrow (A \rightarrow (B \rightarrow C)) \]

Let us generalize this to the dependent types \(\Sigma\) and \(\Pi\). The functional takes a function \(f\) that maps \(z : \Sigma(A, B)\) to \(C(z)\); the resulting function maps \(x \in A\) and \(y \in B(x)\) to \(C((x, y))\).
Formally, there are three typing premises. \( A \) is a type; \( B \) is an \( A \)-indexed family of types; \( C \) is a family of types indexed by \( \Sigma(A, B) \). The goal is expressed using \( \text{PROD} f \) to ensure that the parameter corresponding to the functional’s argument is really called \( f \); Isabelle echoes the type using \( \rightarrow \) because there is no explicit dependence upon \( f \).

```
val prems = Goal
  "[| A type; !!x. x:A ==> B(x) type; 
    !!z. z: (SUM x:A. B(x)) ==> C(z) type |
  ] ==> ?a : \( \text{PROD} f: (\text{PROD} z: (\text{SUM} x:A \cdot B(x)). C(z)). 
  \) (\( \text{PROD} x:A \cdot \text{PROD} y:B(x). C(<x,y>) \))";
```

This is a chance to demonstrate \texttt{intr_tac}. Here, the tactic repeatedly applies \( \Pi \)-introduction and proves the rather tiresome typing conditions.

```
by (intr_tac prems);
```

```
Level 0
?a : (\( \text{PROD} z: \text{SUM} x:A. B(x). C(z) \)) -->
  (\( \text{PROD} x:A. \text{PROD} y:B(x). C(<x,y>) \))
1. ?a : (\( \text{PROD} z: \text{SUM} x:A. B(x). C(z) \)) -->
  (\( \text{PROD} x:A. \text{PROD} y:B(x). C(<x,y>) \))
val prems = ["A type [A type]",
  "?x : A ==> B(?x) type [!!x. x : A ==> B(x) type]",
  "?z : SUM x:A. B(x) ==> C(?z) type
  [!!z. z : SUM x:A. B(x) ==> C(z) type"] : thm list
```

```
Using \( \Pi \)-elimination, we solve subgoal 1 by applying the function \( f \).
```

```
by (eresolve_tac [ProdE] 1);
```

```
Level 2
\( \text{lam x xa xb. ?b7(x,xa,xb)} \)
  : (\( \text{PROD} z: \text{SUM} x:A. B(x). C(z) \)) -->(\( \text{PROD} x:A. \text{PROD} y:B(x). C(<x,y>) \))
1. !!f x y.
   [\( f \ : \text{PROD} z: \text{SUM} x:A. B(x). C(z); x : A; y : B(x) \)] ==>
   ?b7(f,x,y) : C(<x,y>)
```

Finally, we verify that the argument’s type is suitable for the function application. This is straightforward using introduction rules.
5.12 Example: proving the Axiom of Choice

Suppose we have a function \( h \in \prod_{x \in A} \sum_{y \in B(x)} C(x, y) \), which takes \( x \in A \) to some \( y \in B(x) \) paired with some \( z \in C(x, y) \). Interpreting propositions as types, this asserts that for all \( x \in A \) there exists \( y \in B(x) \) such that \( C(x, y) \). The Axiom of Choice asserts that we can construct a function \( f \in \prod_{x \in A} B(x) \) such that \( C(x, f'x) \) for all \( x \in A \), where the latter property is witnessed by a function \( g \in \prod_{x \in A} C(x, f'x) \).

In principle, the Axiom of Choice is simple to derive in Constructive Type Theory. The following definitions work:

\[
\begin{align*}
f & \equiv \text{fst} \circ h \\
g & \equiv \text{snd} \circ h
\end{align*}
\]

But a completely formal proof is hard to find. The rules can be applied in countless ways, yielding many higher-order unifiers. The proof can get bogged down in the details. But with a careful selection of derived rules (recall Fig. 5.10) and the type-checking tactics, we can prove the theorem in nine steps.

First, \texttt{intr_tac} applies introduction rules and performs routine type-checking. This instantiates \( ?a \) to a construction involving a \( \lambda \)-abstraction.
and an ordered pair. The pair’s components are themselves \( \lambda \)-abstractions and there is a subgoal for each.

\[
\begin{align*}
& \text{by (intr_tac prems);} \\
& \text{Level 1} \\
& \text{lam x. <lam xa. ?b7(x,xa),lam xa. ?b8(x,xa)>} \\
& : (\text{PROD } x:A. \text{ SUM } y:B(x). \text{ C}(x,y)) \rightarrow \\
& (\text{SUM } f:\text{PROD } x:A. \text{ B}(x)). \text{ PROD } x:A. \text{ C}(x,f \ x)) \\
& 1. \ \forall x. \\
& \text{[| h : PROD } x:A. \text{ SUM } y:B(x). \text{ C}(x,y); \ x : A |] \implies \\
& \ ?b7(h,x) : B(x) \\
& 2. \ \forall x. \\
& \text{[| h : PROD } x:A. \text{ SUM } y:B(x). \text{ C}(x,y); \ x : A |] \implies \\
& \ ?b8(h,x) : C(x,(\text{lam x. ?b7(h,x)} \ x))
\end{align*}
\]

Subgoal 1 asks to find the choice function itself, taking \( x \in A \) to some \( ?b7(h,x) \in B(x) \). Subgoal 2 asks, given \( x \in A \), for a proof object \( ?b8(h,x) \) to witness that the choice function’s argument and result lie in the relation \( C \).

This latter task will take up most of the proof.

\[
\begin{align*}
& \text{by (eresolve_tac [ProdE RS SumE_fst] 1);} \\
& \text{Level 2} \\
& \text{lam x. <lam xa. fst(x \ xa),lam xa. ?b8(x,xa)>} \\
& : (\text{PROD } x:A. \text{ SUM } y:B(x). \text{ C}(x,y)) \rightarrow \\
& (\text{SUM } f:\text{PROD } x:A. \text{ B}(x)). \text{ PROD } x:A. \text{ C}(x,f \ x)) \\
& 1. \ \forall x. \ x : A \implies \ x : A \\
& 2. \ \forall x. \\
& \text{[| h : PROD } x:A. \text{ SUM } y:B(x). \text{ C}(x,y); \ x : A |] \implies \\
& \ ?b8(h,x) : C(x,(\text{lam x. fst(h \ x)}) \ x)
\end{align*}
\]

Above, we have composed \( \text{fst} \) with the function \( h \). Unification has deduced that the function must be applied to \( x \in A \). We have our choice function.

\[
\begin{align*}
& \text{by (assume_tac 1);} \\
& \text{Level 3} \\
& \text{lam x. <lam xa. fst(x \ xa),lam xa. ?b8(x,xa)>} \\
& : (\text{PROD } x:A. \text{ SUM } y:B(x). \text{ C}(x,y)) \rightarrow \\
& (\text{SUM } f:\text{PROD } x:A. \text{ B}(x)). \text{ PROD } x:A. \text{ C}(x,f \ x)) \\
& 1. \ \forall h. \\
& \text{[| h : PROD } x:A. \text{ SUM } y:B(x). \text{ C}(x,y); \ x : A |] \implies \\
& \ ?b8(h,x) : C(x,(\text{lam x. fst(h \ x)}) \ x)
\end{align*}
\]

Before we can compose \( \text{snd} \) with \( h \), the arguments of \( C \) must be simplified. The derived rule \texttt{replace_type} lets us replace a type by any equivalent type, shown below as the schematic term \( ?A_{133}(h,x) \):

\[
\begin{align*}
& \text{by (resolve_tac [replace_type] 1);} \\
& \text{Level 4} \\
& \text{lam x. <lam xa. fst(x \ xa),lam xa. ?b8(x,xa)>} \\
& : (\text{PROD } x:A. \text{ SUM } y:B(x). \text{ C}(x,y)) \rightarrow \\
& (\text{SUM } f:\text{PROD } x:A. \text{ B}(x)). \text{ PROD } x:A. \text{ C}(x,f \ x))
\end{align*}
\]
CHAPTER 5. CONSTRUCTIVE TYPE THEORY

1. \[ \forall h \ x. \ [\ h : \text{PROD} \ x:A. \ \text{SUM} \ y:B(x). \ C(x,y); \ x : A \ ] \Rightarrow \ C(x,(\lambda m. \ \text{fst}(h \ ' x)) \ ' x) = ?A13(h,x) \]

2. \[ \forall h \ x. \ [\ h : \text{PROD} \ x:A. \ \text{SUM} \ y:B(x). \ C(x,y); \ x : A \ ] \Rightarrow \ ?b8(h,x) : ?A13(h,x) \]

The derived rule subst_eqtyparg lets us simplify a type’s argument (by currying, \( C(x) \) is a unary type operator):

\[
\text{by (resolve_tac [subst_eqtyparg] 1);}
\]

Level 5

\[
\text{lam x. <lam xa. \text{fst}(x \ ' xa), lam xa. \ ?b8(x,xa)> : (\text{PROD} x:A. \ \text{SUM} \ y:B(x). \ C(x,y)) \Rightarrow (\text{SUM} f: \text{PROD} x:A. \ B(x). \ \text{PROD} x:A. \ C(x,f \ ' x))}
\]

1. \[ \forall h \ x. \ [\ h : \text{PROD} \ x:A. \ \text{SUM} \ y:B(x). \ C(x,y); \ x : A \ ] \Rightarrow (\lambda m. \ \text{fst}(h \ ' x)) ' x = ?c14(h,x) : ?A14(h,x) \]

2. \[ \forall h \ x \ z. \ [\ h : \text{PROD} \ x:A. \ \text{SUM} \ y:B(x). \ C(x,y); \ x : A; \ z : ?A14(h,x) \ ] \Rightarrow C(x,z) \text{ type} \]

3. \[ \forall h \ x. \ [\ h : \text{PROD} \ x:A. \ \text{SUM} \ y:B(x). \ C(x,y); \ x : A \ ] \Rightarrow \ ?b8(h,x) : C(x,?c14(h,x)) \]

Subgoal 1 requires simply \( \beta \)-contraction, which is the rule ProdC. The term \( ?c14(h,x) \) in the last subgoal receives the contracted result.

\[
\text{by (resolve_tac [ProdC] 1);}
\]

Level 6

\[
\text{lam x. <lam xa. \text{fst}(x \ ' xa), lam xa. \ ?b8(x,xa)> : (\text{PROD} x:A. \ \text{SUM} \ y:B(x). \ C(x,y)) \Rightarrow (\text{SUM} f: \text{PROD} x:A. \ B(x). \ \text{PROD} x:A. \ C(x,f \ ' x))}
\]

1. \[ \forall h \ x. \ [\ h : \text{PROD} \ x:A. \ \text{SUM} \ y:B(x). \ C(x,y); \ x : A \ ] \Rightarrow x : ?A15(h,x) \]

2. \[ \forall h \ x \ xa. \ [\ h : \text{PROD} \ x:A. \ \text{SUM} \ y:B(x). \ C(x,y); \ x : A; \ xa : ?A15(h,x) \ ] \Rightarrow \ \text{fst}(h \ ' xa) : ?B15(h,x,xa) \]

3. \[ \forall h \ x \ z. \ [\ h : \text{PROD} \ x:A. \ \text{SUM} \ y:B(x). \ C(x,y); \ x : A; \ z : ?B15(h,x,x) \ ] \Rightarrow C(x,z) \text{ type} \]

4. \[ \forall h \ x. \ [\ h : \text{PROD} \ x:A. \ \text{SUM} \ y:B(x). \ C(x,y); \ x : A \ ] \Rightarrow \ ?b8(h,x) : C(x,\text{fst}(h \ ' x)) \]

Routine type-checking goals proliferate in Constructive Type Theory, but
typechk_tac quickly solves them. Note the inclusion of SumE_fst along with the premises.

by (typechk_tac (SumE_fst::prems));
Level 7
lam x. <lam xa. fst(x ' xa),lam xa. ?b8(x,xa)>
: (PROD x:A. SUM y:B(x). C(x,y)) -->
(SUM f:PROD x:A. B(x). PROD x:A. C(x,f ' x))
1. !!h x.
   [| h : PROD x:A. SUM y:B(x). C(x,y); x : A |] ==> ?b8(h,x) : C(x,fst(h ' x))

We are finally ready to compose snd with h.

by (eresolve_tac [ProdE RS SumE_snd] 1);
Level 8
lam x. <lam xa. fst(x ' xa),lam xa. snd(x ' xa)>
: (PROD x:A. SUM y:B(x). C(x,y)) -->
(SUM f:PROD x:A. B(x). PROD x:A. C(x,f ' x))
1. !!h x. x : A ==> x : A
2. !!h x. x : A ==> B(x) type
3. !!h x xa. [| x : A; xa : B(x) |] ==> C(x,xa) type

The proof object has reached its final form. We call typechk_tac to finish the type-checking.

by (typechk_tac prems);
Level 9
lam x. <lam xa. fst(x ' xa),lam xa. snd(x ' xa)>
: (PROD x:A. SUM y:B(x). C(x,y)) -->
(SUM f:PROD x:A. B(x). PROD x:A. C(x,f ' x))
No subgoals!

It might be instructive to compare this proof with Martin-Löf’s forward proof of the Axiom of Choice [9, page 50].


BIBLIOGRAPHY


Index

![ symbol, 6, 9, 16, 17, 30
| symbol, 6, 45
|! symbol, 76
[] symbol, 30
# symbol, 30
## symbol, 76
#+ symbol, 76
& symbol, 6, 45
* symbol, 5, 27, 66
* type, 24
+ symbol, 5, 27, 66
+ type, 24
- symbol, 5, 27, 76
--> symbol, 6, 45, 66
: symbol, 15
< constant, 26
< symbol, 27
<-> symbol, 45
<= constant, 26
<= symbol, 15
= symbol, 6, 45, 66
? symbol, 6, 9, 17
?! symbol, 6
@ symbol, 6, 30
 other symbol, 5
' symbol, 66
`` symbol, 15
{ } symbol, 15

0 constant, 8, 27, 64

absdiff_def theorem, 76
absolute difference, 76
add_assoc theorem, 76
add_commute theorem, 76
add_def theorem, 76
add_inverse_diff theorem, 76
add_mp_tac, 75
add_mult_dist theorem, 76
add_safes, 57
add_ttyping theorem, 76
add_unsafes, 57
addC0 theorem, 76
addC_succ theorem, 76
Addssplits, 23
addsplits, 23, 29, 36
ALL symbol, 6, 16, 17, 45
All constant, 6, 45
All_def theorem, 11
all_dupE theorem, 13
allE theorem, 13
allI theorem, 13
allL theorem, 48, 57
allL_thin theorem, 49
allR theorem, 48
and_def theorem, 11
arg_cong theorem, 12
Arith theory, 75
assumptions
in CTT, 63, 74
Ball constant, 15, 17
Ball_def theorem, 17
ballE theorem, 18
ballI theorem, 18
basic theorem, 47
basic_defs, 72
best_tac, 58
Bex constant, 15, 17
Bex_def theorem, 17
<table>
<thead>
<tr>
<th>Index Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bexCI theorem</td>
<td>18, 20</td>
</tr>
<tr>
<td>bxE theorem</td>
<td>18</td>
</tr>
<tr>
<td>bexI theorem</td>
<td>18, 20</td>
</tr>
<tr>
<td>bool type</td>
<td>5</td>
</tr>
<tr>
<td>box_equal theorem</td>
<td>11, 12</td>
</tr>
<tr>
<td>bspec theorem</td>
<td>18</td>
</tr>
<tr>
<td>butlast constant</td>
<td>30</td>
</tr>
<tr>
<td>case symbol</td>
<td>28, 29, 36</td>
</tr>
<tr>
<td>case_sum constant</td>
<td>26</td>
</tr>
<tr>
<td>case_sum_Inl theorem</td>
<td>26</td>
</tr>
<tr>
<td>case_sum_Inr theorem</td>
<td>26</td>
</tr>
<tr>
<td>case_tac</td>
<td>11, 37</td>
</tr>
<tr>
<td>case_weak_cong</td>
<td>36</td>
</tr>
<tr>
<td>Collect constant</td>
<td>14, 15</td>
</tr>
<tr>
<td>Collect_mem_eq theorem</td>
<td>17</td>
</tr>
<tr>
<td>Collect theorem</td>
<td>18, 42</td>
</tr>
<tr>
<td>CollectE theorem</td>
<td>18</td>
</tr>
<tr>
<td>CollectI theorem</td>
<td>18, 43</td>
</tr>
<tr>
<td>comp_rls</td>
<td>72</td>
</tr>
<tr>
<td>Compl_def theorem</td>
<td>17</td>
</tr>
<tr>
<td>Compl_disjoint theorem</td>
<td>21</td>
</tr>
<tr>
<td>Compl_Int theorem</td>
<td>21</td>
</tr>
<tr>
<td>Compl_partition theorem</td>
<td>21</td>
</tr>
<tr>
<td>Compl_Un theorem</td>
<td>21</td>
</tr>
<tr>
<td>ComplD theorem</td>
<td>19</td>
</tr>
<tr>
<td>ComplI theorem</td>
<td>19</td>
</tr>
<tr>
<td>concat constant</td>
<td>30</td>
</tr>
<tr>
<td>cong theorem</td>
<td>12</td>
</tr>
<tr>
<td>conj_cong theorem</td>
<td>22</td>
</tr>
<tr>
<td>conjE theorem</td>
<td>12</td>
</tr>
<tr>
<td>conjI theorem</td>
<td>12</td>
</tr>
<tr>
<td>conjL theorem</td>
<td>48</td>
</tr>
<tr>
<td>conjR theorem</td>
<td>48</td>
</tr>
<tr>
<td>conjunct1 theorem</td>
<td>12</td>
</tr>
<tr>
<td>conjunct2 theorem</td>
<td>12</td>
</tr>
<tr>
<td>Constructive Type Theory</td>
<td>63–86</td>
</tr>
<tr>
<td>context</td>
<td>43</td>
</tr>
<tr>
<td>contL theorem</td>
<td>49</td>
</tr>
<tr>
<td>contLS theorem</td>
<td>47</td>
</tr>
<tr>
<td>contr theorem</td>
<td>49</td>
</tr>
<tr>
<td>contr constant</td>
<td>64</td>
</tr>
<tr>
<td>contRS theorem</td>
<td>47</td>
</tr>
<tr>
<td>could_res</td>
<td>52</td>
</tr>
<tr>
<td>could_resolve_seq</td>
<td>52</td>
</tr>
<tr>
<td>CTT theory</td>
<td>1, 63</td>
</tr>
<tr>
<td>Cube theory</td>
<td>1</td>
</tr>
<tr>
<td>cut theorem</td>
<td>47</td>
</tr>
<tr>
<td>cutL_tac</td>
<td>51</td>
</tr>
<tr>
<td>cutR_tac</td>
<td>51</td>
</tr>
<tr>
<td>datatype</td>
<td>29</td>
</tr>
<tr>
<td>Delsplits</td>
<td>23</td>
</tr>
<tr>
<td>diff_0_eq_0 theorem</td>
<td>76</td>
</tr>
<tr>
<td>diff_def</td>
<td>76</td>
</tr>
<tr>
<td>diff_self_eq_0 theorem</td>
<td>76</td>
</tr>
<tr>
<td>diff_succ_succ theorem</td>
<td>76</td>
</tr>
<tr>
<td>diff_typing theorem</td>
<td>76</td>
</tr>
<tr>
<td>diffC0 theorem</td>
<td>76</td>
</tr>
<tr>
<td>disjCI theorem</td>
<td>13</td>
</tr>
<tr>
<td>disjE theorem</td>
<td>12</td>
</tr>
<tr>
<td>disjI1 theorem</td>
<td>12</td>
</tr>
<tr>
<td>disjI2 theorem</td>
<td>12</td>
</tr>
<tr>
<td>disjL theorem</td>
<td>48</td>
</tr>
<tr>
<td>disjR theorem</td>
<td>48</td>
</tr>
<tr>
<td>div symbol</td>
<td>27, 76</td>
</tr>
<tr>
<td>div_def</td>
<td>76</td>
</tr>
<tr>
<td>div_geq</td>
<td>27</td>
</tr>
<tr>
<td>div_less</td>
<td>27</td>
</tr>
<tr>
<td>Divides theory</td>
<td>26</td>
</tr>
<tr>
<td>double_complement theorem</td>
<td>21</td>
</tr>
<tr>
<td>drop constant</td>
<td>30</td>
</tr>
<tr>
<td>dropWhile constant</td>
<td>30</td>
</tr>
<tr>
<td>dvd symbol</td>
<td>27</td>
</tr>
<tr>
<td>Elem constant</td>
<td>64</td>
</tr>
<tr>
<td>elim_rls</td>
<td>72</td>
</tr>
<tr>
<td>elimL_rls</td>
<td>72</td>
</tr>
</tbody>
</table>
empty_def theorem, 17
empty_pack, 56
emptyE theorem, 19
Eps constant, 6, 8
Eq constant, 64
eq constant, 64, 72
EqC theorem, 71
EqE theorem, 71
Eqelem constant, 64
EqF theorem, 71
EqFL theorem, 71
EqI theorem, 71
Eqtype constant, 64
equal_tac, 73
equal_types theorem, 67
equal_typesL theorem, 67
equalityCE theorem, 18, 20, 42, 43
equalityD1 theorem, 18
equalityD2 theorem, 18
equalityE theorem, 18
equalityI theorem, 18
EX symbol, 6, 16, 17, 45
Ex constant, 6, 45
EX! symbol, 6
Ex1 constant, 6
Ex1_def theorem, 11
ex1E theorem, 13
ex1I theorem, 13
Ex_def theorem, 11
exCI theorem, 13
excluded_middle theorem, 13
exE theorem, 13
exI theorem, 13
exL theorem, 48
exR theorem, 48, 52, 57
exR_thin theorem, 49, 52, 53
ext theorem, 10
F constant, 64
False constant, 6, 45
False_def theorem, 11
FalseE theorem, 12
FalseL theorem, 48
fast_tac, 58
FE theorem, 70, 75
FEL theorem, 70
FF theorem, 70
filseq_resolve_tac, 52
filt_resolve_tac, 52, 73
filter constant, 30
flex-flex constraints, 55
FOL theory, 74
foldl constant, 30
form_rls, 72
formL_rls, 72
forms_of_seq, 51
fst constant, 24, 64, 71, 72
fst_conv theorem, 24
fst_def theorem, 69
Fun theory, 21
fun type, 5
fun_cong theorem, 12
function applications
  in CTT, 66
hd constant, 30
higher-order logic, 5–43
HOL, 9
HOL system, 5, 9
HOL_basic_ss, 22
HOL_ss, 22
HOLCF theory, 1
hyp_rew_tac, 74
hyp_subst_tac, 22
i type, 63
If constant, 6
if, 22
if_def theorem, 11
if_not_P theorem, 13
if_P theorem, 13
if_weak_cong, 22
INDEX

Least constant, 6
Least_def theorem, 11
length constant, 30
less_induct theorem, 28
Let constant, 6, 9
let symbol, 7
Let_def theorem, 9, 11
Lin_Arith.tac, 29
linorder class, 8, 26, 28
List theory, 29, 30
list type, 29
list.split theorem, 29
LK theory, 1, 44, 49
LK_dup_pack, 57, 58
LK_pack, 57
map constant, 30
max constant, 8, 26
mem symbol, 30
mem_Collect_eq theorem, 17
min constant, 8, 26
minus class, 5
mod symbol, 27, 76
mod_def theorem, 76
mod_geq theorem, 27
mod_less theorem, 27
Modal theory, 1
mono constant, 8
mp theorem, 10
mp_tac, 74
mult_assoc theorem, 76
mult_commute theorem, 76
mult_def theorem, 76
mult_typing theorem, 76
multC0 theorem, 76
multC_succ theorem, 76
N constant, 64
n_not_Suc_n theorem, 27
Nat theory, 26
nat type, 26, 27
nat theorem, 8
nat_induct theorem, 27
NatArith theory, 26
NCO theorem, 68
NC_succ theory, 68
NE theorem, 68, 71, 77
NEL theorem, 68
NF theorem, 68, 78
NIO theorem, 68
NI_succ theory, 68
NI_SuccL theorem, 68
NIO theorem, 77
Not constant, 6, 45
not_def theorem, 11
not_sym theorem, 12
notE theorem, 12
notI theorem, 12
notL theorem, 48
notnotD theorem, 13
notR theorem, 48
null constant, 30
o type, 44
o symbol, 6, 22
o_def theorem, 11
of symbol, 9
or_def theorem, 11
Ord theory, 8
ord class, 8, 9, 26
order class, 8
pack, 56
pack ML type, 56
pack_of thy, 56
Pair constant, 24
pair constant, 64
Pair_inject theorem, 24
pc_tac, 58, 75, 81, 83
plus class, 5
plus_ac0 class, 8
<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>PlusC_inl theorem, 70</td>
<td>94</td>
</tr>
<tr>
<td>PlusC_inr theorem, 70</td>
<td>94</td>
</tr>
<tr>
<td>PlusE theorem, 70, 75, 79</td>
<td>94</td>
</tr>
<tr>
<td>PlusEL theorem, 70</td>
<td>94</td>
</tr>
<tr>
<td>PlusF theorem, 70</td>
<td>94</td>
</tr>
<tr>
<td>PlusFL theorem, 70</td>
<td>94</td>
</tr>
<tr>
<td>PlusI_inl theorem, 70, 80</td>
<td>94</td>
</tr>
<tr>
<td>PlusI_inrL theorem, 70</td>
<td>94</td>
</tr>
<tr>
<td>Pow constant, 15</td>
<td>94</td>
</tr>
<tr>
<td>Pow_def theorem, 17</td>
<td>94</td>
</tr>
<tr>
<td>PowD theorem, 19</td>
<td>94</td>
</tr>
<tr>
<td>power class, 5</td>
<td>94</td>
</tr>
<tr>
<td>Primrec symbol, 28</td>
<td>94</td>
</tr>
<tr>
<td>Priorities, 3</td>
<td>94</td>
</tr>
<tr>
<td>Prod symbol, 65, 66</td>
<td>94</td>
</tr>
<tr>
<td>Prod constant, 64</td>
<td>94</td>
</tr>
<tr>
<td>Prod theory, 24</td>
<td>94</td>
</tr>
<tr>
<td>prod.exhaust theorem, 24</td>
<td>94</td>
</tr>
<tr>
<td>prod.inject theorem, 24</td>
<td>94</td>
</tr>
<tr>
<td>prod.split theorem, 24</td>
<td>94</td>
</tr>
<tr>
<td>ProdC theorem, 68, 85</td>
<td>94</td>
</tr>
<tr>
<td>ProdC2 theorem, 68</td>
<td>94</td>
</tr>
<tr>
<td>ProdE theorem, 68, 82, 84, 86</td>
<td>94</td>
</tr>
<tr>
<td>ProdEL theorem, 68</td>
<td>94</td>
</tr>
<tr>
<td>ProdF theorem, 68</td>
<td>94</td>
</tr>
<tr>
<td>ProdFL theorem, 68</td>
<td>94</td>
</tr>
<tr>
<td>ProdI theorem, 68, 75, 77</td>
<td>94</td>
</tr>
<tr>
<td>ProdIL theorem, 68</td>
<td>94</td>
</tr>
<tr>
<td>Prop_pack, 57</td>
<td>94</td>
</tr>
<tr>
<td>Range constant, 15, 42</td>
<td>94</td>
</tr>
<tr>
<td>Range_def theorem, 17</td>
<td>94</td>
</tr>
<tr>
<td>RangeE theorem, 19, 42</td>
<td>94</td>
</tr>
<tr>
<td>RealInl theorem, 19</td>
<td>94</td>
</tr>
<tr>
<td>Real theorem, 8, 28</td>
<td>94</td>
</tr>
<tr>
<td>Rec constant, 64, 71</td>
<td>94</td>
</tr>
<tr>
<td>Rec_nat constant, 28</td>
<td>94</td>
</tr>
<tr>
<td>Recdef, 38–41</td>
<td>94</td>
</tr>
<tr>
<td>Recursion</td>
<td>94</td>
</tr>
<tr>
<td>General, 38–41</td>
<td>94</td>
</tr>
<tr>
<td>Red_if_equal theorem, 67</td>
<td>94</td>
</tr>
<tr>
<td>Reduce constant, 64, 69, 74</td>
<td>94</td>
</tr>
<tr>
<td>Refl theorem, 10, 47</td>
<td>94</td>
</tr>
<tr>
<td>Refl_elem theorem, 67, 72</td>
<td>94</td>
</tr>
<tr>
<td>Refl_red theorem, 67</td>
<td>94</td>
</tr>
<tr>
<td>Refl_type theorem, 67, 72</td>
<td>94</td>
</tr>
<tr>
<td>REPEAT_FIRST, 73</td>
<td>94</td>
</tr>
<tr>
<td>Repeat_goal_tac, 58</td>
<td>94</td>
</tr>
<tr>
<td>Replace_type theorem, 71, 84</td>
<td>94</td>
</tr>
<tr>
<td>Res_inst_tac, 58</td>
<td>94</td>
</tr>
<tr>
<td>Rew_tac, 74</td>
<td>94</td>
</tr>
<tr>
<td>Rl, 79</td>
<td>94</td>
</tr>
<tr>
<td>Rs, 84, 86</td>
<td>94</td>
</tr>
<tr>
<td>Safe_goal_tac, 58</td>
<td>94</td>
</tr>
<tr>
<td>Safe_tac, 75</td>
<td>94</td>
</tr>
<tr>
<td>Safestep_tac, 75</td>
<td>94</td>
</tr>
<tr>
<td>Search</td>
<td>94</td>
</tr>
<tr>
<td>Best-first, 43</td>
<td>94</td>
</tr>
<tr>
<td>Seqof constant, 45</td>
<td>94</td>
</tr>
<tr>
<td>Sequent calculus, 44–58</td>
<td>94</td>
</tr>
<tr>
<td>Set theory, 14, 17</td>
<td>94</td>
</tr>
<tr>
<td>Set constant, 30</td>
<td>94</td>
</tr>
<tr>
<td>Set type, 14</td>
<td>94</td>
</tr>
<tr>
<td>Set_diff_def theorem, 17</td>
<td>94</td>
</tr>
<tr>
<td>Show_sorts, 8</td>
<td>94</td>
</tr>
<tr>
<td>Show_types, 8</td>
<td>94</td>
</tr>
<tr>
<td>Sigma constant, 24</td>
<td>94</td>
</tr>
<tr>
<td>Sigma_def theorem, 24</td>
<td>94</td>
</tr>
<tr>
<td>SigmaE theorem, 24</td>
<td>94</td>
</tr>
<tr>
<td>SigmaI theorem, 24</td>
<td>94</td>
</tr>
<tr>
<td>Simplification</td>
<td>94</td>
</tr>
<tr>
<td>Of case, 36</td>
<td>94</td>
</tr>
<tr>
<td>Of if, 22</td>
<td>94</td>
</tr>
<tr>
<td>Of conjunctions, 22</td>
<td>94</td>
</tr>
<tr>
<td>Size constant, 37</td>
<td>94</td>
</tr>
<tr>
<td>Smp_tac, 11</td>
<td>94</td>
</tr>
<tr>
<td>Index Item</td>
<td>Page Numbers</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>snd constant</td>
<td>24, 64, 71, 72</td>
</tr>
<tr>
<td>snd_conv theorem</td>
<td>24</td>
</tr>
<tr>
<td>snd_def theorem</td>
<td>69</td>
</tr>
<tr>
<td>subj type</td>
<td>44</td>
</tr>
<tr>
<td>SOME symbol</td>
<td>6</td>
</tr>
<tr>
<td>some_equality</td>
<td>10, 13</td>
</tr>
<tr>
<td>someI theorem</td>
<td>10</td>
</tr>
<tr>
<td>spec theorem</td>
<td>13</td>
</tr>
<tr>
<td>split constant</td>
<td>24, 64, 79</td>
</tr>
<tr>
<td>split theorem</td>
<td>24</td>
</tr>
<tr>
<td>split_all_tac</td>
<td>25</td>
</tr>
<tr>
<td>split_if theorem</td>
<td>13, 23</td>
</tr>
<tr>
<td>ssubst theorem</td>
<td>11, 12</td>
</tr>
<tr>
<td>stac</td>
<td>22</td>
</tr>
<tr>
<td>step_tac</td>
<td>58, 75</td>
</tr>
<tr>
<td>strip_tac</td>
<td>11</td>
</tr>
<tr>
<td>subset_def theorem</td>
<td>17</td>
</tr>
<tr>
<td>subset_refl theorem</td>
<td>18</td>
</tr>
<tr>
<td>subset_trans</td>
<td>18</td>
</tr>
<tr>
<td>subsetCE theorem</td>
<td>18, 20</td>
</tr>
<tr>
<td>subsetD theorem</td>
<td>18, 20</td>
</tr>
<tr>
<td>subsetI theorem</td>
<td>18</td>
</tr>
<tr>
<td>subst theorem</td>
<td>10, 47</td>
</tr>
<tr>
<td>subst_elem</td>
<td>67</td>
</tr>
<tr>
<td>subst_elemL</td>
<td>67</td>
</tr>
<tr>
<td>subst_eqtyparg</td>
<td>71, 85</td>
</tr>
<tr>
<td>subst_prodE</td>
<td>71, 72</td>
</tr>
<tr>
<td>subst_type</td>
<td>67</td>
</tr>
<tr>
<td>subst_typeL</td>
<td>67</td>
</tr>
<tr>
<td>Suc constant</td>
<td>27</td>
</tr>
<tr>
<td>Suc_not_Zero</td>
<td>27</td>
</tr>
<tr>
<td>succ constant</td>
<td>64</td>
</tr>
<tr>
<td>Sum symbol</td>
<td>65, 66</td>
</tr>
<tr>
<td>Sum constant</td>
<td>64</td>
</tr>
<tr>
<td>Sum theory</td>
<td>25</td>
</tr>
<tr>
<td>sum constant</td>
<td>8</td>
</tr>
<tr>
<td>sum_split_case</td>
<td>26</td>
</tr>
<tr>
<td>SumC theorem</td>
<td>69</td>
</tr>
<tr>
<td>SumE theorem</td>
<td>69, 75, 79</td>
</tr>
<tr>
<td>sumE theorem</td>
<td>26</td>
</tr>
<tr>
<td>SumE_fst theorem</td>
<td>71, 72, 84, 86</td>
</tr>
<tr>
<td>SumE_snd theorem</td>
<td>71, 72, 86</td>
</tr>
<tr>
<td>SumEL theorem</td>
<td>69</td>
</tr>
<tr>
<td>SumF theorem</td>
<td>69</td>
</tr>
<tr>
<td>SumFL theorem</td>
<td>69</td>
</tr>
<tr>
<td>SumI theorem</td>
<td>69, 80</td>
</tr>
<tr>
<td>SumIL theorem</td>
<td>69</td>
</tr>
<tr>
<td>SumIL2 theorem</td>
<td>71</td>
</tr>
<tr>
<td>surj constant</td>
<td>21, 22</td>
</tr>
<tr>
<td>surj_def</td>
<td>21</td>
</tr>
<tr>
<td>surjective_pairing</td>
<td>24</td>
</tr>
<tr>
<td>surjective_sum</td>
<td>26</td>
</tr>
<tr>
<td>swap theorem</td>
<td>13</td>
</tr>
<tr>
<td>swap_res_tac</td>
<td>43</td>
</tr>
<tr>
<td>sym theorem</td>
<td>12</td>
</tr>
<tr>
<td>sym_elem</td>
<td>67</td>
</tr>
<tr>
<td>sym_type</td>
<td>67</td>
</tr>
<tr>
<td>symL theorem</td>
<td>49</td>
</tr>
<tr>
<td>symR theorem</td>
<td>49</td>
</tr>
<tr>
<td>T constant</td>
<td>64</td>
</tr>
<tr>
<td>t type</td>
<td>63</td>
</tr>
<tr>
<td>take constant</td>
<td>30</td>
</tr>
<tr>
<td>takeWhile constant</td>
<td>30</td>
</tr>
<tr>
<td>TC theorem</td>
<td>70</td>
</tr>
<tr>
<td>TE theorem</td>
<td>70</td>
</tr>
<tr>
<td>TEL theorem</td>
<td>70</td>
</tr>
<tr>
<td>term class</td>
<td>5, 44</td>
</tr>
<tr>
<td>test_assume_tac</td>
<td>73</td>
</tr>
<tr>
<td>TF theorem</td>
<td>70</td>
</tr>
<tr>
<td>THE symbol</td>
<td>45</td>
</tr>
<tr>
<td>The constant</td>
<td>45</td>
</tr>
<tr>
<td>The theorem</td>
<td>48</td>
</tr>
<tr>
<td>the_equality</td>
<td>49</td>
</tr>
<tr>
<td>thinL theorem</td>
<td>49</td>
</tr>
<tr>
<td>thinLS theorem</td>
<td>47</td>
</tr>
<tr>
<td>thinR theorem</td>
<td>49</td>
</tr>
<tr>
<td>thinRS theorem</td>
<td>47</td>
</tr>
<tr>
<td>TI theorem</td>
<td>70</td>
</tr>
<tr>
<td>times class</td>
<td>5</td>
</tr>
<tr>
<td>tl constant</td>
<td>30</td>
</tr>
<tr>
<td>tracing</td>
<td></td>
</tr>
</tbody>
</table>
INDEX

of unification, 8
trans theorem, 12
trans_elem theorem, 67
trans_red theorem, 67
trans_type theorem, 67
transR theorem, 49
True constant, 6, 45
True_def theorem, 11, 48
True_or_False theorem, 10
TrueI theorem, 12
Trueprop constant, 6, 45
TrueR theorem, 49
tt constant, 64
Type constant, 64
typechk_tac, 73, 78, 81, 86
UN symbol, 15–17
Un symbol, 15
Un1 theorem, 20
Un2 theorem, 20
Un_absorb theorem, 21
Un_assoc theorem, 21
Un_commute theorem, 21
Un_def theorem, 17
UN_E theorem, 19
UN_I theorem, 19
Un_Int_distrib theorem, 21
Un_Inter theorem, 21
Un_least theorem, 20
Un_upper1 theorem, 20
Un_upper2 theorem, 20
UnCI theorem, 19, 20
UnE theorem, 19
UnI1 theorem, 19
UnI2 theorem, 19
unification
   incompleteness of, 8
Unify.trace_types, 8
UNION constant, 15
Union constant, 15
UNION1 constant, 15
UNION1_def theorem, 17
UNION_def theorem, 17
Union_def theorem, 17
Union_least theorem, 20
Union_Un_distrib theorem, 21
Union_upper theorem, 20
UnionE theorem, 19
UnionI theorem, 19
unit_eq theorem, 25
when constant, 64, 71, 79
zero_ne_succ theorem, 68, 69