Isabelle's Logics

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

With Contributions by Tobias Nipkow and Markus Wenzel[1]

8 October 2017

[1] Markus 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

Preface

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 $S43$.

`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/
http://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 $(\ldots)$ express grouping
- constructs followed by a Kleene star, such as id^* and $(\ldots)^*$ 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 $\exists$ has lower priority than $\lor$, which has lower priority than $\land$. There, $P \land Q \lor R$ abbreviates $(P \land Q) \lor R$ rather than $P \land (Q \lor R)$. Also, $\exists x\, .\, P \lor Q$ abbreviates $\exists x\, .\, (P \lor Q)$ rather than $(\exists x\, .\, P) \lor Q$. Note especially that $P \lor (\exists x\, .\, Q)$ becomes syntactically invalid if the brackets are removed.

A **binder** is a symbol associated with a constant of type $(\sigma \Rightarrow \tau) \Rightarrow \tau'$. For instance, we may declare $\forall$ as a binder for the constant All, which has type $(\alpha \Rightarrow o) \Rightarrow o$. This defines the syntax $\forall x\, .\, t$ to mean $All(\lambda x\, .\, t)$. We can also write $\forall x_1 \ldots x_m\, .\, t$ to abbreviate $\forall x_1\, .\, \ldots \forall x_m\, .\, t$; this is possible for any constant provided that τ and τ' are the same type. The Hilbert description operator $\varepsilon x\, .\, P\, x$ has type $(\alpha \Rightarrow bool) \Rightarrow \alpha$ 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`~=`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 σ and τ 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 `+`, `-`, `*`. and `^`. They are overloaded to denote

name	*meta-type*	*description*
`Trueprop`	$bool \Rightarrow prop$	coercion to *prop*
`Not`	$bool \Rightarrow bool$	negation ($\neg$)
`True`	$bool$	tautology ($\top$)
`False`	$bool$	absurdity ($\bot$)
`If`	$[bool, \alpha, \alpha] \Rightarrow \alpha$	conditional
`Let`	$[\alpha, \alpha \Rightarrow \beta] \Rightarrow \beta$	let binder

Constants

symbol	*name*	*meta-type*	*description*
`SOME` or `@`	`Eps`	$(\alpha \Rightarrow bool) \Rightarrow \alpha$	Hilbert description (ε)
`ALL` or `!`	`All`	$(\alpha \Rightarrow bool) \Rightarrow bool$	universal quantifier ($\forall$)
`EX` or `?`	`Ex`	$(\alpha \Rightarrow bool) \Rightarrow bool$	existential quantifier ($\exists$)
`EX!` or `?!`	`Ex1`	$(\alpha \Rightarrow bool) \Rightarrow bool$	unique existence ($\exists!$)
`LEAST`	`Least`	$(\alpha :: ord \Rightarrow bool) \Rightarrow \alpha$	least element

Binders

symbol	*meta-type*	*priority*	*description*
`o`	$[\beta \Rightarrow \gamma, \alpha \Rightarrow \beta] \Rightarrow (\alpha \Rightarrow \gamma)$	Left 55	composition ($\circ$)
`=`	$[\alpha, \alpha] \Rightarrow bool$	Left 50	equality ($=$)
`<`	$[\alpha :: ord, \alpha] \Rightarrow bool$	Left 50	less than ($<$)
`<=`	$[\alpha :: ord, \alpha] \Rightarrow bool$	Left 50	less than or equals ($\leq$)
`&`	$[bool, bool] \Rightarrow bool$	Right 35	conjunction ($\wedge$)
`|`	$[bool, bool] \Rightarrow bool$	Right 30	disjunction ($\vee$)
`-->`	$[bool, bool] \Rightarrow bool$	Right 25	implication ($\rightarrow$)

Infixes

Figure 2.1: Syntax of `HOL`

```
   term  =  expression of class term
         |  SOME id . formula        |  @ id . formula
         |  let id = term; ... ; id = term in term
         |  if formula then term else term
         |  LEAST id . formula

formula  =  expression of type bool
         |  term = term
         |  term ~= term
         |  term < term
         |  term <= term
         |  ~ formula
         |  formula & formula
         |  formula | formula
         |  formula --> formula
         |  ALL id id* . formula     |  !   id id* . formula
         |  EX  id id* . formula     |  ?   id id* . formula
         |  EX! id id* . formula     |  ?!  id id* . formula
```

Figure 2.2: Full grammar for HOL

the obvious arithmetic operations on types `nat`, `int` and `real`. (With the `^` operator, the exponent always has type `nat`.) Non-arithmetic overloadings are also done: the operator `-` can denote set difference, while `^` 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 `0` is also overloaded. It serves as the zero element of several types, of which the most important is `nat` (the natural numbers). The type class `plus_ac0` comprises all types for which 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 `nat`, `int` and `real` and also multisets. The summation operator `sum` is available for all types in this class.

Theory `Ord` defines the syntactic class `ord` of order signatures. The relations $<$ and $\leq$ are polymorphic over this class, as are the functions `mono`, `min` and `max`, and the `LEAST` operator. `Ord` also defines a subclass `order` of `ord` which axiomatizes the types that are partially ordered with respect to $\leq$. A further subclass `linorder` of `order` axiomatizes linear orderings. For details, see the file `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 and k in the goal $i \leq j \Longrightarrow i \leq j+k$ have type $\alpha :: \{ord, plus\}$, although you may have expected them to have some numeric type, e.g. *nat*. Instead you should have stated the goal as $(i :: nat) \leq j \Longrightarrow i \leq j+k$, which causes all three variables to have type *nat*.

! If resolution fails for no obvious reason, try setting `show_types` to `true`, causing Isabelle to display types of terms. Possibly set `show_sorts` to `true` as well, causing Isabelle to display type classes and sorts.

Where function types are involved, Isabelle's unification code does not guarantee to find instantiations for type variables automatically. Be prepared to use `res_inst_tac` instead of `resolve_tac`, possibly instantiating type variables. Setting `Unify.trace_types` to `true` causes Isabelle to report omitted search paths during unification.

2.1.2 Binders

Hilbert's **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 `Eps`$(\lambda x \,.\, P[x])$ or use the syntax `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 `ALL` and `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 HOL` to the `isabelle` executable), then `!` and `?` are displayed.

If τ is a type of class `ord`, P a formula and x a variable of type τ, then the term `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 ε choice operator, so `Least` is always meaningful, but may yield nothing useful in case there is not a unique least element satisfying P.[1]

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 `let` construct whose syntax appears in Fig. 2.2. Internally it is translated into the constant `Let`. It can be expanded by rewriting with its definition, `Let_def`.

HOL also defines the basic syntax

$$\texttt{case}\ e\ \texttt{of}\ c_1\ \texttt{=>}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ c_n\ \texttt{=>}\ e_n$$

as a uniform means of expressing `case` constructs. Therefore `case` and `of` are reserved words. Initially, this is mere syntax and has no logical meaning. By declaring translations, you can cause instances of the `case` construct to denote applications of particular case operators. This is what happens automatically for each `datatype` definition (see §2.6).

[1] Class *ord* does not require much of its instances, so $\leq$ need not be a well-ordering, not even an order at all!

```
refl           t = (t::'a)
subst          [| s = t; P s |] ==> P (t::'a)
ext            (!!x::'a. (f x :: 'b) = g x) ==> (%x. f x) = (%x. g x)
impI           (P ==> Q) ==> P-->Q
mp             [| P-->Q;  P |] ==> Q
iff            (P-->Q) --> (Q-->P) --> (P=Q)
someI          P(x::'a) ==> P(@x. P x)
True_or_False  (P=True) | (P=False)
```

Figure 2.3: The `HOL` rules

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 $f\ x = \texttt{if}\ldots\texttt{then}\ldots\texttt{else}\ldots$ you need to write $f\ x = (\texttt{if}\ldots\texttt{then}\ldots\texttt{else}\ldots)$.

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 ε-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 (`==`) for definitions.

[2] In fact, the ε-operator already makes the logic classical, as shown by Diaconescu; see Paulson [14] for details.

```
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 $\neg P$, 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 $\forall\vec{x}\,.\,P\vec{x}\rightarrow Q\vec{x}$ and $P\vec{a}$, ($\vec{x}$ being a vector of j variables) then it replaces the universally quantified implication by $Q\vec{a}$. It may instantiate unknowns. It fails if it can do nothing.

```
sym          s=t ==> t=s
trans        [| r=s; s=t |] ==> r=t
ssubst       [| t=s; P s |] ==> P t
box_equals   [| a=b;  a=c;  b=d |] ==> c=d
arg_cong     x = y ==> f x = f y
fun_cong     f = g ==> f x = g x
cong         [| f = g; x = y |] ==> f x = g y
not_sym      t ~= s ==> s ~= t
```

Equality

```
TrueI        True
FalseE       False ==> P

conjI        [| P; Q |] ==> P&Q
conjunct1    [| P&Q |] ==> P
conjunct2    [| P&Q |] ==> Q
conjE        [| P&Q;  [| P; Q |] ==> R |] ==> R

disjI1       P ==> P|Q
disjI2       Q ==> P|Q
disjE        [| P | Q; P ==> R; Q ==> R |] ==> R

notI         (P ==> False) ==> ~ P
notE         [| ~ P;  P |] ==> R
impE         [| P-->Q;  P;  Q ==> R |] ==> R
```

Propositional logic

```
iffI         [| P ==> Q;  Q ==> P |] ==> P=Q
iffD1        [| P=Q; P |] ==> Q
iffD2        [| P=Q; Q |] ==> P
iffE         [| P=Q; [| P --> Q; Q --> P |] ==> R |] ==> R
```

Logical equivalence

Figure 2.5: Derived rules for HOL

```
allI      (!!x. P x) ==> !x. P x
spec      !x. P x ==> P x
allE      [| !x. P x;  P x ==> R |] ==> R
all_dupE  [| !x. P x;  [| P x; !x. P x |] ==> R |] ==> R

exI       P x ==> ? x. P x
exE       [| ? x. P x; !!x. P x ==> Q |] ==> Q

ex1I      [| P a;  !!x. P x ==> x=a |] ==> ?! x. P x
ex1E      [| ?! x. P x;  !!x. [| P x;  ! y. P y --> y=x |] ==> R
          |] ==> R

some_equality   [| P a;  !!x. P x ==> x=a |] ==> (@x. P x) = a
```

Quantifiers and descriptions

```
ccontr          (~P ==> False) ==> P
classical       (~P ==> P) ==> P
excluded_middle ~P | P

disjCI       (~Q ==> P) ==> P|Q
exCI         (! x. ~ P x ==> P a) ==> ? x. P x
impCE        [| P-->Q; ~ P ==> R; Q ==> R |] ==> R
iffCE        [| P=Q;  [| P;Q |] ==> R;  [| ~P; ~Q |] ==> R |] ==> R
notnotD      ~~P ==> P
swap         ~P ==> (~Q ==> P) ==> Q
```

Classical logic

```
if_P         P ==> (if P then x else y) = x
if_not_P     ~ P ==> (if P then x else y) = y
split_if     P(if Q then x else y) = ((Q --> P x) & (~Q --> P y))
```

Conditionals

Figure 2.6: More derived rules

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 σ. 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 $\alpha \Rightarrow bool$. 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 $\alpha \Rightarrow bool$ 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 $\{a_1, \ldots\}$ notation abbreviates finite sets constructed in the obvious manner using `insert` and $\{\}$:

$$\{a, b, c\} \quad \equiv \quad \texttt{insert}\, a\, (\texttt{insert}\, b\, (\texttt{insert}\, c\, \{\}))$$

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 $\texttt{Collect}(\lambda x\,.\,P[x])$. 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	$(\alpha \Rightarrow bool) \Rightarrow \alpha\,set$	comprehension
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	$[\alpha\,set, \alpha \Rightarrow bool] \Rightarrow bool$	bounded quantifiers

Constants

symbol	*name*	*meta-type*	*priority*	*description*
INT	INTER1	$(\alpha \Rightarrow \beta\,set) \Rightarrow \beta\,set$	10	intersection
UN	UNION1	$(\alpha \Rightarrow \beta\,set) \Rightarrow \beta\,set$	10	union

Binders

symbol	*meta-type*	*priority*	*description*
‘‘	$[\alpha \Rightarrow \beta, \alpha\,set] \Rightarrow \beta\,set$	Left 90	image
Int	$[\alpha\,set, \alpha\,set] \Rightarrow \alpha\,set$	Left 70	intersection ($\cap$)
Un	$[\alpha\,set, \alpha\,set] \Rightarrow \alpha\,set$	Left 65	union ($\cup$)
:	$[\alpha, \alpha\,set] \Rightarrow bool$	Left 50	membership ($\in$)
<=	$[\alpha\,set, \alpha\,set] \Rightarrow bool$	Left 50	subset ($\subseteq$)

Infixes

Figure 2.7: Syntax of the theory Set

external	*internal*	*description*
a ~: b	~(a : b)	not in
{a_1, ...}	insert a_1 ... {}	finite set
{x. $P[x]$}	Collect($\lambda x . P[x]$)	comprehension
INT $x{:}A$. $B[x]$	INTER A $\lambda x . B[x]$	intersection
UN $x{:}A$. $B[x]$	UNION A $\lambda x . B[x]$	union
ALL $x{:}A$. $P[x]$ or ! $x{:}A$. $P[x]$	Ball A λx. $P[x]$	bounded ∀
EX $x{:}A$. $P[x]$ or ? $x{:}A$. $P[x]$	Bex A λx. $P[x]$	bounded ∃

Translations

$$\begin{array}{rcl}
\mathit{term} & = & \text{other terms}\ldots \\
 & | & \texttt{\{\}} \\
 & | & \texttt{\{}\ \mathit{term}\ (\texttt{,}\mathit{term})^*\ \texttt{\}} \\
 & | & \texttt{\{}\ \mathit{id}\ \ \texttt{.}\ \mathit{formula}\ \texttt{\}} \\
 & | & \mathit{term}\ \texttt{`{}`}\ \mathit{term} \\
 & | & \mathit{term}\ \texttt{Int}\ \mathit{term} \\
 & | & \mathit{term}\ \texttt{Un}\ \mathit{term} \\
 & | & \texttt{INT}\ \ \mathit{id}{:}\mathit{term}\ \ \texttt{.}\ \mathit{term} \\
 & | & \texttt{UN}\ \ \ \mathit{id}{:}\mathit{term}\ \ \texttt{.}\ \mathit{term} \\
 & | & \texttt{INT}\ \ \mathit{id}\ \mathit{id}^*\ \ \texttt{.}\ \mathit{term} \\
 & | & \texttt{UN}\ \ \ \mathit{id}\ \mathit{id}^*\ \ \texttt{.}\ \mathit{term} \\
 & & \\
\mathit{formula} & = & \text{other formulae}\ldots \\
 & | & \mathit{term}\ \ \texttt{:}\ \ \mathit{term} \\
 & | & \mathit{term}\ \ \texttt{\~{}:}\ \ \mathit{term} \\
 & | & \mathit{term}\ \ \texttt{<=}\ \ \mathit{term} \\
 & | & \texttt{ALL}\ \mathit{id}{:}\mathit{term}\ \ \texttt{.}\ \mathit{formula}\ \ |\ \ \texttt{!}\ \mathit{id}{:}\mathit{term}\ \ \texttt{.}\ \mathit{formula} \\
 & | & \texttt{EX}\ \ \mathit{id}{:}\mathit{term}\ \ \texttt{.}\ \mathit{formula}\ \ |\ \ \texttt{?}\ \mathit{id}{:}\mathit{term}\ \ \texttt{.}\ \mathit{formula}
\end{array}$$

Full Grammar

Figure 2.8: Syntax of the theory Set (continued)

```
mem_Collect_eq     (a : {x. P x}) = P a
Collect_mem_eq     {x. x:A} = A

empty_def          {}           == {x. False}
insert_def         insert a B   == {x. x=a} Un B
Ball_def           Ball A P     == ! x. x:A --> P x
Bex_def            Bex A P      == ? x. x:A & P x
subset_def         A <= B       == ! x:A. x:B
Un_def             A Un B       == {x. x:A | x:B}
Int_def            A Int B      == {x. x:A & x:B}
set_diff_def       A - B        == {x. x:A & x~:B}
Compl_def          -A           == {x. ~ x:A}
INTER_def          INTER A B    == {y. ! x:A. y: B x}
UNION_def          UNION A B    == {y. ? x:A. y: B x}
INTER1_def         INTER1 B     == INTER {x. True} B
UNION1_def         UNION1 B     == UNION {x. True} B
Inter_def          Inter S      == (INT x:S. x)
Union_def          Union S      == (UN  x:S. x)
Pow_def            Pow A        == {B. B <= A}
image_def          f``A         == {y. ? x:A. y=f x}
range_def          range f      == {y. ? x. y=f x}
```

Figure 2.9: Rules of the theory `Set`

The set theory defines two **bounded quantifiers**:

$$\begin{array}{lcl} \forall x\in A\,.\,P[x] & \text{abbreviates} & \forall x\,.\,x\in A\to P[x] \\ \exists x\in A\,.\,P[x] & \text{abbreviates} & \exists x\,.\,x\in A\land P[x] \end{array}$$

The constants `Ball` and `Bex` are defined accordingly. Instead of `Ball` A P and `Bex` A P we may write `ALL` $x\,{:}\,A$`.` $P[x]$ and `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 `UN` $x\,{:}\,A$`.` $B[x]$ and `INT` $x\,{:}\,A$`.` $B[x]$.

Unions and intersections over types, namely $\bigcup_x B[x]$ and $\bigcap_x B[x]$, are written `UN` x`.` $B[x]$ and `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 `mem_Collect_eq` and `Collect_mem_eq` assert that the functions `Collect` and `op :` are isomorphisms. Of course, `op :` also serves as the membership relation.

```
CollectI        [| P a |] ==> a : {x. P x}
CollectD        [| a : {x. P x} |] ==> P a
CollectE        [| a : {x. P x};  P a ==> W |] ==> W

ballI           [| !!x. x:A ==> P x |] ==> ! x:A. P x
bspec           [| ! x:A. P x;  x:A |] ==> P x
ballE           [| ! x:A. P x;  P x ==> Q;  ~ x:A ==> Q |] ==> Q

bexI            [| P x;  x:A |] ==> ? x:A. P x
bexCI           [| ! x:A. ~ P x ==> P a;  a:A |] ==> ? x:A. P x
bexE            [| ? x:A. P x;  !!x. [| x:A; P x |] ==> Q  |] ==> Q
```

Comprehension and Bounded quantifiers

```
subsetI         (!!x. x:A ==> x:B) ==> A <= B
subsetD         [| A <= B;  c:A |] ==> c:B
subsetCE        [| A <= B;  ~ (c:A) ==> P;  c:B ==> P |] ==> P

subset_refl     A <= A
subset_trans    [| A<=B;  B<=C |] ==> A<=C

equalityI       [| A <= B;  B <= A |] ==> A = B
equalityD1      A = B ==> A<=B
equalityD2      A = B ==> B<=A
equalityE       [| A = B;  [| A<=B; B<=A |] ==> P |]  ==>  P

equalityCE      [| A = B;  [| c:A; c:B |] ==> P;
                           [| ~ c:A; ~ c:B |] ==> P
                |]  ==>  P
```

The subset and equality relations

Figure 2.10: Derived rules for set theory

```
emptyE   a : {} ==> P

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

ComplI   [| c:A ==> False |] ==> c : -A
ComplD   [| c : -A |] ==> ~ c:A

UnI1     c:A ==> c : A Un B
UnI2     c:B ==> c : A Un B
UnCI     (~c:B ==> c:A) ==> c : A Un B
UnE      [| c : A Un B;  c:A ==> P;  c:B ==> P |] ==> P

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

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

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

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

InterI   [| !!X. X:C ==> A:X |] ==> A : Inter C
InterD   [| A : Inter C;  X:C |] ==> A:X
InterE   [| A : Inter C;  A:X ==> R;  ~ X:C ==> R |] ==> R

PowI     A<=B ==> A: Pow B
PowD     A: Pow B ==> A<=B

imageI   [| x:A |] ==> f x : f‘‘A
imageE   [| b : f‘‘A;  !!x.[| b=f x;  x:A |] ==> P |] ==> P

rangeI   f x : range f
rangeE   [| b : range f;  !!x.[| b=f x |] ==> P |] ==> P
```

Figure 2.11: Further derived rules for set theory

```
Union_upper      B:A ==> B <= Union A
Union_least      [| !!X. X:A ==> X<=C |] ==> Union A <= C

Inter_lower      B:A ==> Inter A <= B
Inter_greatest   [| !!X. X:A ==> C<=X |] ==> C <= Inter A

Un_upper1        A <= A Un B
Un_upper2        B <= A Un B
Un_least         [| A<=C;  B<=C |] ==> A Un B <= C

Int_lower1       A Int B <= A
Int_lower2       A Int B <= B
Int_greatest     [| C<=A;  C<=B |] ==> C <= A Int B
```

Figure 2.12: Derived rules involving subsets

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

```
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

name	*meta-type*	*description*
`inj` `surj`	$(\alpha \Rightarrow \beta) \Rightarrow bool$	injective/surjective
`inj_on`	$[\alpha \Rightarrow \beta, \alpha\, set] \Rightarrow bool$	injective over subset
`inv`	$(\alpha \Rightarrow \beta) \Rightarrow (\beta \Rightarrow \alpha)$	inverse function

```
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 o) and the predicate surj is done simply by expanding the definitions.

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

2.4 Simplification and substitution

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

See the *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 conj_cong in a simpset, addcongs [conj_cong].

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

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.

HOL provides the tactic hyp_subst_tac, which substitutes for an equality throughout a subgoal and its hypotheses. This tactic uses 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`:

$$?P(\texttt{if }?b\texttt{ then }?x\texttt{ else }?y) \;=\; ((?b \to ?P(?x)) \land (\neg ?b \to ?P(?y))) \qquad (*)$$

For example, a simple instance of $(*)$ is

$$x \in (\texttt{if }x \in A\texttt{ then }A\texttt{ else }\{x\}) \;=\; ((x \in A \to x \in A) \land (x \notin A \to x \in \{x\}))$$

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 acording 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

$$?P(c\ ?x_1\ \ldots\ ?x_n) \;=\; rhs$$

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

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

```
Addsplits: 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 α *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

symbol	*meta-type*	*description*
`Pair`	$[\alpha,\beta] \Rightarrow \alpha\times\beta$	ordered pairs (a,b)
`fst`	$\alpha\times\beta \Rightarrow \alpha$	first projection
`snd`	$\alpha\times\beta \Rightarrow \beta$	second projection
`split`	$[[\alpha,\beta] \Rightarrow \gamma, \alpha\times\beta] \Rightarrow \gamma$	generalized projection
`Sigma`	$[\alpha\, set, \alpha \Rightarrow \beta\, set] \Rightarrow (\alpha\times\beta) set$	general sum of sets

```
Sigma_def    Sigma A B == UN x:A. UN y:B x. {(x,y)}

prod.inject      ((a,b) = (a’,b’)) = (a=a’ & b=b’)
Pair_inject  [| (a, b) = (a’,b’);  [| a=a’;  b=b’ |] ==> R |] ==> R
prod.exhaust        [| !!x y. p = (x,y) ==> Q |] ==> Q

fst_conv     fst (a,b) = a
snd_conv     snd (a,b) = b
surjective_pairing  p = (fst p,snd p)

split        case_prod c (a,b) = c a b
prod.split  R(case_prod c p) = (! x y. p = (x,y) --> R(c x y))

SigmaI     [| a:A;  b:B a |] ==> (a,b) : Sigma A B

SigmaE     [| c:Sigma A B; !!x y.[| x:A; y:B x; c=(x,y) |] ==> P
           |] ==> P
```

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

Theory `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:

external	internal
$\tau_1 \times \ldots \times \tau_n$	$\tau_1 \times (\ldots(\tau_{n-1}\times\tau_n)\ldots)$
$(t_1,\ldots,t_n)$	$(t_1,(\ldots,(t_{n-1},t_n)\ldots)$

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

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

Nested patterns are also supported. They are translated stepwise:

$$\begin{array}{rcl}
\texttt{\%(}x\texttt{,}y\texttt{,}z\texttt{).}\ t & \leadsto & \texttt{\%(}x\texttt{,(}y\texttt{,}z\texttt{)).}\ t \\
 & \leadsto & \texttt{case\_prod(\%}x\texttt{.\%(}y\texttt{,}z\texttt{).}\ t\texttt{)} \\
 & \leadsto & \texttt{case\_prod(\%}x\texttt{.}\ \texttt{case\_prod(\%}y\ \ z\texttt{.}\ t\texttt{))}
\end{array}$$

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 λ-abstractions, patterns can be used in any variable binding construct which is internally described by a λ-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

```
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*
`Inl`	$\alpha \Rightarrow \alpha + \beta$	first injection
`Inr`	$\beta \Rightarrow \alpha + \beta$	second injection
`case_sum`	$[\alpha \Rightarrow \gamma, \beta \Rightarrow \gamma, \alpha + \beta] \Rightarrow \gamma$	conditional

```
Inl_not_Inr    Inl a ~= Inr b

inj_Inl        inj Inl
inj_Inr        inj Inr

sumE           [| !!x. P(Inl x);  !!y. P(Inr y) |] ==> P s

case_sum_Inl   case_sum f g (Inl x) = f x
case_sum_Inr   case_sum f g (Inr x) = g x

surjective_sum case_sum (%x. f(Inl x)) (%y. f(Inr y)) s = f s
sum.split_case R(case_sum f g s) = ((! x. s = Inl(x) --> R(f(x))) &
                                    (! y. s = Inr(y) --> R(g(y))))
```

Figure 2.16: Type $\alpha + \beta$

2.5.2 The type of natural numbers, *nat*

The theory `Nat` defines the natural numbers in a roundabout but traditional way. The axiom of infinity postulates a type *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 *nat* is an instance of class `ord`, which makes the overloaded functions of this class (especially `<` and `<=`, but also `min`, `max` and `LEAST`) available on *nat*. Theory `Nat` also shows that `<=` is a linear order, so *nat* is also an instance of class `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 `+`, `-` and `*` on `nat` are part of the default simpset.

Functions on *nat* can be defined by primitive or well-founded recursion; see §2.7. A simple example is addition. Here, `op +` is the name of the infix operator `+`, following the standard convention.

symbol	*meta-type*	*priority*	*description*
0	α		zero
Suc	$nat \Rightarrow nat$		successor function
*	$[\alpha, \alpha] \Rightarrow \alpha$	Left 70	multiplication
div	$[\alpha, \alpha] \Rightarrow \alpha$	Left 70	division
mod	$[\alpha, \alpha] \Rightarrow \alpha$	Left 70	modulus
dvd	$[\alpha, \alpha] \Rightarrow bool$	Left 70	"divides" relation
+	$[\alpha, \alpha] \Rightarrow \alpha$	Left 65	addition
-	$[\alpha, \alpha] \Rightarrow \alpha$	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

```
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 $sd \ldots d$, where s is an optional minus sign and $d \ldots d$ 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 sufficent, you can invoke the full procedure `Lin_Arith.tac` explicitly. It copes with arbitrary formulae involving `=`, `<`, `<=`, `+`, `-`, `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 `min`, `max`, and `-` because they require case distinctions. If `k` is a numeral, then `div k`, `mod k` and `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, `Lin_Arith.tac` may take super-exponential time and space.

If `Lin_Arith.tac` fails, try to find relevant arithmetic results in the library. The theories `Nat` and `NatArith` contain theorems about `<`, `<=`, `+`, `-` and `*`. Theory `Divides` contains theorems about `div` and `mod`. Use Proof General's *find* button (or other search facilities) to locate them.

2.5.4 The type constructor for lists, *list*

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

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

is defined by translation. For details see §2.6. There is also a case splitting rule `list.split`

$$\begin{array}{l} P(\texttt{case}\ e\ \texttt{of}\ \texttt{[]}\ \texttt{=>}\ a\ \texttt{|}\ x\texttt{\#}xs\ \texttt{=>}\ f\ x\ xs) \ = \\ ((e = \texttt{[]} \to P(a)) \land (\forall x\ xs\ .\ e = x\texttt{\#}xs \to P(f\ x\ xs))) \end{array}$$

which can be fed to `addsplits` just like `split_if` (see §2.4.1).

`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 ML, frequently appear in applications of Isabelle/HOL. In principle, such types could be defined by hand via `typedef`, but this would be far too tedious. The `datatype` definition

symbol	*meta-type*	*priority*	*description*
`[]`	$\alpha\ list$		empty list
`#`	$[\alpha, \alpha\ list] \Rightarrow \alpha\ list$	Right 65	list constructor
`null`	$\alpha\ list \Rightarrow bool$		emptiness test
`hd`	$\alpha\ list \Rightarrow \alpha$		head
`tl`	$\alpha\ list \Rightarrow \alpha\ list$		tail
`last`	$\alpha\ list \Rightarrow \alpha$		last element
`butlast`	$\alpha\ list \Rightarrow \alpha\ list$		drop last element
`@`	$[\alpha\ list, \alpha\ list] \Rightarrow \alpha\ list$	Left 65	append
`map`	$(\alpha \Rightarrow \beta) \Rightarrow (\alpha\ list \Rightarrow \beta\ list)$		apply to all
`filter`	$(\alpha \Rightarrow bool) \Rightarrow (\alpha\ list \Rightarrow \alpha\ list)$		filter functional
`set`	$\alpha\ list \Rightarrow \alpha\ set$		elements
`mem`	$\alpha \Rightarrow \alpha\ list \Rightarrow bool$	Left 55	membership
`foldl`	$(\beta \Rightarrow \alpha \Rightarrow \beta) \Rightarrow \beta \Rightarrow \alpha\ list \Rightarrow \beta$		iteration
`concat`	$(\alpha\ list)list \Rightarrow \alpha\ list$		concatenation
`rev`	$\alpha\ list \Rightarrow \alpha\ list$		reverse
`length`	$\alpha\ list \Rightarrow nat$		length
`!`	$\alpha\ list \Rightarrow nat \Rightarrow \alpha$	Left 100	indexing
`take`, `drop`	$nat \Rightarrow \alpha\ list \Rightarrow \alpha\ list$		take/drop a prefix
`takeWhile`, `dropWhile`	$(\alpha \Rightarrow bool) \Rightarrow \alpha\ list \Rightarrow \alpha\ list$		take/drop a prefix

Constants and infixes

external	*internal*	*description*
$[x_1,\ \ldots,\ x_n]$	x_1 `#` $\cdots$ `#` x_n `#` `[]`	finite list
$[x\!:\!l.\ P]$	`filter` $(\lambda x.P)$ l	list comprehension

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

package of Isabelle/HOL (cf. [1]) automates such chores. It generates an appropriate `typedef` based on a least fixed-point construction, and proves freeness theorems and induction rules, as well as theorems for recursion and case combinators. The user just has to give a simple specification of new inductive types using a notation similar to ML or Haskell.

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:

$$\begin{array}{llcl} \texttt{datatype} & (\vec{\alpha})t_1 & = & C_1^1\ \tau_{1,1}^1\ \ldots\ \tau_{1,m_1^1}^1\ \mid\ \ldots\ \mid\ C_{k_1}^1\ \tau_{k_1,1}^1\ \ldots\ \tau_{k_1,m_{k_1}^1}^1 \\ & & \vdots & \\ \texttt{and} & (\vec{\alpha})t_n & = & C_1^n\ \tau_{1,1}^n\ \ldots\ \tau_{1,m_1^n}^n\ \mid\ \ldots\ \mid\ C_{k_n}^n\ \tau_{k_n,1}^n\ \ldots\ \tau_{k_n,m_{k_n}^n}^n \end{array}$$

where $\vec{\alpha} = (\alpha_1, \ldots, \alpha_h)$ is a list of type variables, C_i^j are distinct constructor names and $\tau_{i,i'}^j$ are *admissible* types containing at most the type variables $\alpha_1, \ldots, \alpha_h$. A type τ occurring in a `datatype` definition is *admissible* if and only if

- τ is non-recursive, i.e. τ 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
- $\tau = (\tau'_1, \ldots, \tau'_{h'})t'$, where t' is the type constructor of an already existing datatype and $\tau'_1, \ldots, \tau'_{h'}$ are admissible types.
- $\tau = \sigma \to \tau'$, where τ' is an admissible type and σ 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 $\tau^j_{i,i'}$ of the form

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

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

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

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

```
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

```
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.

```
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_i with the following property: for all argument types $\tau^j_{i,i'}$ 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_i^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 `list` is a non-empty datatype, `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:

$$C_i^j :: [\tau_{i,1}^j, \ldots, \tau_{i,m_i^j}^j] \Rightarrow (\alpha_1, \ldots, \alpha_h)t_j$$

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

$$C_i^j\ x_1\ \ldots\ x_{m_i^j} \neq C_{i'}^j\ y_1\ \ldots\ y_{m_{i'}^j} \qquad \text{for all } i \neq i'.$$

The constructor functions are injective:

$$(C_i^j\ x_1\ \ldots\ x_{m_i^j} = C_i^j\ y_1\ \ldots\ y_{m_i^j}) = (x_1 = y_1 \wedge \ldots \wedge x_{m_i^j} = y_{m_i^j})$$

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 `DatatypePackage.distinct_simproc`.

Structural induction

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

$$\begin{array}{c}
\begin{array}{lcl}
\bigwedge x_1 \ldots x_{m_1^1} \,.\, \llbracket P_{s_{1,1}^1}\ x_{r_{1,1}^1}; \ldots ; P_{s^1_{1,l_1^1}}\ x_{r^1_{1,l_1^1}} \rrbracket & \Longrightarrow & P_1\ \left(C_1^1\ x_1\ \ldots\ x_{m_1^1} \right) \\
 & \vdots & \\
\bigwedge x_1 \ldots x_{m_{k_1}^1} \,.\, \llbracket P_{s_{k_1,1}^1}\ x_{r_{k_1,1}^1}; \ldots ; P_{s^1_{k_1,l_{k_1}^1}}\ x_{r^1_{k_1,l_{k_1}^1}} \rrbracket & \Longrightarrow & P_1\ \left(C_{k_1}^1\ x_1\ \ldots\ x_{m_{k_1}^1} \right) \\
 & \vdots & \\
\bigwedge x_1 \ldots x_{m_1^n} \,.\, \llbracket P_{s_{1,1}^n}\ x_{r_{1,1}^n}; \ldots ; P_{s^n_{1,l_1^n}}\ x_{r^n_{1,l_1^n}} \rrbracket & \Longrightarrow & P_n\ \left(C_1^n\ x_1\ \ldots\ x_{m_1^n} \right) \\
 & \vdots & \\
\bigwedge x_1 \ldots x_{m_{k_n}^n} \,.\, \llbracket P_{s_{k_n,1}^n}\ x_{r_{k_n,1}^n}; \ldots P_{s^n_{k_n,l_{k_n}^n}}\ x_{r^n_{k_n,l_{k_n}^n}} \rrbracket & \Longrightarrow & P_n\ \left(C_{k_n}^n\ x_1\ \ldots\ x_{m_{k_n}^n} \right)
\end{array} \\
\hline
P_1\ x_1 \wedge \ldots \wedge P_n\ x_n
\end{array}$$

where

$$\begin{array}{rcl}
Rec_i^j & := & \left\{ \left(r_{i,1}^j, s_{i,1}^j \right), \ldots, \left(r^j_{i,l_i^j}, s^j_{i,l_i^j} \right) \right\} = \\
 & & \left\{ (i', i'') \;\middle|\; 1 \leq i' \leq m_i^j \wedge 1 \leq i'' \leq n \wedge \tau_{i,i'}^j = (\alpha_1, \ldots, \alpha_h) t_{i''} \right\}
\end{array}$$

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

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

to an equivalent definition without nesting:

```
datatype ('a,'b) term      = Var
                           | App 'b (('a, 'b) term_list)
and      ('a,'b) term_list = Nil'
                           | Cons' (('a,'b) term) (('a,'b) term_list)
```

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

$$\begin{array}{c}
\begin{array}{l}
\bigwedge x \,.\ P_1\ (\mathtt{Var}\ x) \\
\bigwedge x_1\ x_2 \,.\ P_2\ x_2 \Longrightarrow P_1\ (\mathtt{App}\ x_1\ x_2) \\
P_2\ \mathtt{Nil} \\
\bigwedge x_1\ x_2 \,.\, \llbracket P_1\ x_1; P_2\ x_2 \rrbracket \Longrightarrow P_2\ (\mathtt{Cons}\ x_1\ x_2)
\end{array} \\
\hline
P_1\ x_1 \wedge P_2\ x_2
\end{array}$$

Note that there are two predicates P_1 and P_2, one for the type `('a,'b) term` and one for the type `(('a, 'b) term) list`.

For a datatype with function types such as `'a tree`, the induction rule is of the form

$$\frac{\bigwedge a\,.\ P\ (\mathtt{Atom}\ a) \qquad \bigwedge ts\,.\ (\forall x\,.\ P\ (ts\ x)) \Longrightarrow P\ (\mathtt{Branch}\ ts)}{P\ t}$$

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:

$$\begin{array}{rrcl}
\mathtt{case}\ e\ \mathtt{of} & C^j_1\ x_{1,1}\ \ldots\ x_{1,m^j_1} & \Rightarrow & e_1 \\
\vdots & & & \\
\mid & C^j_{k_j}\ x_{k_j,1}\ \ldots\ x_{k_j,m^j_{k_j}} & \Rightarrow & e_{k_j}
\end{array}$$

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`.split` is provided:

$$\begin{array}{rl}
P(t_j\mathtt{\_case}\ f_1\ \ldots\ f_{k_j}\ e) = & ((\forall x_1 \ldots x_{m^j_1}\,.\ e = C^j_1\ x_1 \ldots x_{m^j_1} \to P(f_1\ x_1 \ldots x_{m^j_1})) \\
& \wedge\ \ldots\ \wedge \\
& (\forall x_1 \ldots x_{m^j_{k_j}}\,.\ e = C^j_{k_j}\ x_1 \ldots x_{m^j_{k_j}} \to P(f_{k_j}\ x_1 \ldots x_{m^j_{k_j}})))
\end{array}$$

where t_j`_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`.split_asm` in the same manner. Both rules are available under t_j`.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`.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 $\alpha \Rightarrow nat$. Each datatype defines a particular instance of `size` by overloading according to the following scheme:

$$size(C_i^j\ x_1\ \ldots\ x_{m_i^j}) = \begin{cases} 0 & \text{if } Rec_i^j = \emptyset \\ 1 + \sum\limits_{h=1}^{l_i^j} size\ x_{r_{i,h}^j} & \text{if } Rec_i^j = \left\{\left(r_{i,1}^j, s_{i,1}^j\right), \ldots, \left(r_{i,l_i^j}^j, s_{i,l_i^j}^j\right)\right\} \end{cases}$$

where Rec_i^j 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 "`$x_1\ \ldots\ x_n$`"` 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 $C_1^j, \ldots C_{k_j}^j$, subgoal i is replaced by k_j new subgoals which contain the additional assumption $u = C_{i'}^j\ x_1\ \ldots\ x_{m_{i'}^j}$ for $i' = 1, \ldots, k_j$.

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:

```
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
```

`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_..._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.[4]

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`.

[4] This facility is based on Konrad Slind's TFL package [16]. Thanks are due to Konrad for implementing TFL and assisting with its installation.

```
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 $\prec$ 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 τ, then $\prec$ has to be a relation over τ: 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 $(nat \times nat)set$, while $<$ has type $[nat, nat] \Rightarrow bool$.
- `measure` f, where f has type $\tau \Rightarrow nat$, is the relation $\prec$ on type τ 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`<*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.

```
recdef 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

```
recdef 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 `gcd` shown above, Isabelle/HOL is unable to prove one termination condition. It remains as a precondition of the recursion theorems:

```
gcd.simps;
  ["! m n. n ~= 0 --> m mod n < n
    ==> gcd (?m,?n) = (if ?n=0 then ?m else gcd (?n, ?m mod ?n))"]
  : 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:

```
Tfl.tgoalw thy [] gcd.simps;
  Level 0
  ! m n. n ~= 0 --> m mod n < n
   1. ! m n. n ~= 0 --> m mod n < n
```

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.

```
recdef gcd "measure ((%(m,n). n) ::nat*nat=>nat)"
  simpset "simpset() addsimps [mod_less_divisor, zero_less_eq]"
    "gcd (m, n) = (if n=0 then m else gcd(n, m mod n))"
```

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

```
gcd.induct;
  "(!!m n. n ~= 0 --> ?P n (m mod n) ==> ?P m n) ==> ?P ?u ?v" : thm
```

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:

$$\forall f :: \alpha \Rightarrow \alpha \Rightarrow bool \,.\, \exists S :: \alpha \Rightarrow bool \,.\, \forall x :: \alpha \,.\, f\ x \neq S$$

Viewing types as sets, $\alpha \Rightarrow bool$ represents the powerset of α. This version states that for every function from α to its powerset, some subset is outside its range.

The Isabelle proof uses HOL's set theory, with the type $\alpha\, set$ and the operator `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 `rangeE` replaces $?S \in \mathtt{range}\, f$ 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 `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 `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 $\{x \,.\, x \notin f\ x\}$, 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.

Chapter 3

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 Γ and Δ 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 `term`. No types of individuals are provided, but extensions can define types such as `nat::term` and type constructors such as `list::(term)term`. Below, the type variable α ranges over class `term`; the equality symbol and quantifiers are polymorphic (many-sorted). The type of formulae is o, which belongs to class `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 `Sequents` as the object-logic. Once in Isabelle, change the context to theory `LK.thy`:

```
isabelle Sequents
context LK.thy;
```

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 $sobj \Rightarrow sobj$ represents a list of formulae.

The **definite description** operator $\iota x \,.\, P[x]$ 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 `THE` x`.` $P[x]$. The corresponding rule (Fig. 3.4) does not entail the Axiom of Choice because it requires uniqueness.

name	*meta-type*	*description*
`Trueprop`	$[sobj \Rightarrow sobj, sobj \Rightarrow sobj] \Rightarrow prop$	coercion to *prop*
`Seqof`	$[o, sobj] \Rightarrow sobj$	singleton sequence
`Not`	$o \Rightarrow o$	negation ($\neg$)
`True`	o	tautology ($\top$)
`False`	o	absurdity ($\bot$)

Constants

symbol	*name*	*meta-type*	*priority*	*description*
`ALL`	`All`	$(\alpha \Rightarrow o) \Rightarrow o$	10	universal quantifier ($\forall$)
`EX`	`Ex`	$(\alpha \Rightarrow o) \Rightarrow o$	10	existential quantifier ($\exists$)
`THE`	`The`	$(\alpha \Rightarrow o) \Rightarrow \alpha$	10	definite description (ι)

Binders

symbol	*meta-type*	*priority*	*description*
`=`	$[\alpha, \alpha] \Rightarrow o$	Left 50	equality ($=$)
`&`	$[o, o] \Rightarrow o$	Right 35	conjunction ($\wedge$)
`|`	$[o, o] \Rightarrow o$	Right 30	disjunction ($\vee$)
`-->`	$[o, o] \Rightarrow o$	Right 25	implication ($\rightarrow$)
`<->`	$[o, o] \Rightarrow o$	Right 25	biconditional ($\leftrightarrow$)

Infixes

external	*internal*	*description*
Γ `|-` Δ	`Trueprop(`Γ`, `Δ`)`	sequent $\Gamma \vdash \Delta$

Translations

Figure 3.1: Syntax of `LK`

$$\begin{array}{rcl}
\mathit{prop} & = & \mathit{sequence}\ \texttt{|-}\ \mathit{sequence} \\[1ex]
\mathit{sequence} & = & \mathit{elem}\quad (\texttt{,}\ \ \mathit{elem})^{*} \\
 & | & \mathit{empty} \\[1ex]
\mathit{elem} & = & \texttt{\$}\ \mathit{term} \\
 & | & \mathit{formula} \\
 & | & \texttt{<<}\mathit{sequence}\texttt{>>} \\[1ex]
\mathit{formula} & = & \text{expression of type } o \\
 & | & \mathit{term}\ \texttt{=}\ \mathit{term} \\
 & | & \texttt{\~{}}\ \mathit{formula} \\
 & | & \mathit{formula}\ \texttt{\&}\ \mathit{formula} \\
 & | & \mathit{formula}\ \texttt{|}\ \mathit{formula} \\
 & | & \mathit{formula}\ \texttt{-->}\ \mathit{formula} \\
 & | & \mathit{formula}\ \texttt{<->}\ \mathit{formula} \\
 & | & \texttt{ALL}\ \mathit{id}\ \mathit{id}^{*}\ \ \texttt{.}\ \mathit{formula} \\
 & | & \texttt{EX}\quad \mathit{id}\ \mathit{id}^{*}\ \ \texttt{.}\ \mathit{formula} \\
 & | & \texttt{THE}\ \mathit{id}\quad \texttt{.}\ \mathit{formula}
\end{array}$$

Figure 3.2: Grammar of LK

```
basic       $H, P, $G |- $E, P, $F

contRS      $H |- $E, $S, $S, $F ==> $H |- $E, $S, $F
contLS      $H, $S, $S, $G |- $E ==> $H, $S, $G |- $E

thinRS      $H |- $E, $F ==> $H |- $E, $S, $F
thinLS      $H, $G |- $E ==> $H, $S, $G |- $E

cut         [| $H |- $E, P;  $H, P |- $E |] ==> $H |- $E
```

Structural rules

```
refl        $H |- $E, a=a, $F
subst       $H(a), $G(a) |- $E(a) ==> $H(b), a=b, $G(b) |- $E(b)
```

Equality rules

Figure 3.3: Basic Rules of `LK`

Conditional expressions are available with the notation

`if` *formula* `then` *term* `else` *term*.

Figure 3.2 presents the grammar of LK. Traditionally, Γ and Δ are meta-variables 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 `<<`*sequence*`>>` stands for a sequence of formulæ. For example, you can declare the constant `imps` to consist of two implications:

```
consts     P,Q,R :: o
constdefs imps :: seq'=>seq'
        "imps == <<P --> Q, Q --> R>>"
```

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

```
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
disjL    [| $H, P, $G |- $E;  $H, Q, $G |- $E |] ==> $H, P|Q, $G |- $E

impR     $H, P |- $E, Q, $F ==> $H |- $E, P-->Q, $F
impL     [| $H,$G |- $E,P;  $H, Q, $G |- $E |] ==> $H, P-->Q, $G |- $E

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

```
thinR        $H |- $E, $F ==> $H |- $E, P, $F
thinL        $H, $G |- $E ==> $H, P, $G |- $E

contR        $H |- $E, P, P, $F ==> $H |- $E, P, $F
contL        $H, P, P, $G |- $E ==> $H, P, $G |- $E

symR         $H |- $E, $F, a=b ==> $H |- $E, b=a, $F
symL         $H, $G, b=a |- $E ==> $H, a=b, $G |- $E

transR       [| $H|- $E, $F, a=b;  $H|- $E, $F, b=c |]
             ==> $H|- $E, a=c, $F

TrueR        $H |- $E, True, $F

iffR         [| $H, P |- $E, Q, $F;  $H, Q |- $E, P, $F |]
             ==> $H |- $E, P<->Q, $F

iffL         [| $H, $G |- $E, P, Q;  $H, Q, P, $G |- $E |]
             ==> $H, P<->Q, $G |- $E

allL_thin    $H, P(x), $G |- $E ==> $H, ALL x. P(x), $G |- $E
exR_thin     $H |- $E, P(x), $F ==> $H |- $E, EX x. P(x), $F

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

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 $\leftrightarrow$ is defined using $\land$ and $\to$. 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 $\leftrightarrow$. 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 `res_inst_tac` can instantiate the variable `?P` in these rules, specifying the formula to duplicate. See theory `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 `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 `Simp_tac` refers to the default simpset (`simpset()`). With sequents, the `full_` and `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 `Asm_full_simp_tac`.)

For classical reasoning, several tactics are available:

```
Safe_tac : int -> tactic
Step_tac : int -> tactic
Fast_tac : int -> tactic
Best_tac : int -> tactic
Pc_tac   : int -> 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":

```
Add_safes   : thm list -> unit
Add_unsafes : thm list -> 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:

$$\frac{\Gamma \vdash \Delta, P \land Q}{\Gamma \vdash \Delta, Q \land P}$$

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.

```
cutR_tac : string -> int -> tactic
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.

`cutR_tac` P i reads an LK formula P, and applies the cut rule to subgoal i. It then deletes some formula from the right side of subgoal i, replacing that formula by P.

`cutL_tac` P i reads an LK formula P, and applies the cut rule to subgoal i. It then deletes some formula from the left side of the new subgoal $i+1$, replacing that formula by P.

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

3.4 Tactics for sequents

```
forms_of_seq       : term -> term list
could_res          : term * term -> bool
could_resolve_seq  : term * term -> bool
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.

`forms_of_seq` t returns the list of all formulae in the sequent t, removing sequence variables.

`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 $\vdash \exists y . \forall x . P(y) \to P(x)$ 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 `exR` rather than the weaker `exR_thin`.

```
Goal "|- EX y. ALL x. P(y)-->P(x)";
  Level 0
   |- EX y. ALL x. P(y) --> P(x)
   1.  |- EX y. ALL x. P(y) --> P(x)
by (resolve_tac [exR] 1);
  Level 1
   |- EX y. ALL x. P(y) --> P(x)
   1.  |- ALL x. P(?x) --> P(x), EX x. ALL xa. P(x) --> P(xa)
```

There are now two formulae on the right side. Keeping the existential one in reserve, we break down the universal one.

```
by (resolve_tac [allR] 1);
  Level 2
   |- EX y. ALL x. P(y) --> P(x)
   1. !!x.  |- P(?x) --> P(x), EX x. ALL xa. P(x) --> P(xa)
by (resolve_tac [impR] 1);
  Level 3
   |- EX y. ALL x. P(y) --> P(x)
   1. !!x. P(?x) |- P(x), EX x. ALL xa. P(x) --> P(xa)
```

Because LK is a sequent calculus, the formula $P(?x)$ does not become an 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 $?x$.

```
by (resolve_tac [basic] 1);
  by: tactic failed
```

We reuse the existential formula using `exR_thin`, which discards it; we shall not need it a third time. We again break down the resulting formula.

```
by (resolve_tac [exR_thin] 1);
  Level 4
   |- EX y. ALL x. P(y) --> P(x)
   1. !!x. P(?x) |- P(x), ALL xa. P(?x7(x)) --> P(xa)
by (resolve_tac [allR] 1);
  Level 5
   |- EX y. ALL x. P(y) --> P(x)
   1. !!x xa. P(?x) |- P(x), P(?x7(x)) --> P(xa)
by (resolve_tac [impR] 1);
  Level 6
   |- EX y. ALL x. P(y) --> P(x)
   1. !!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 $?x_7$ to $\lambda x . x$, transforming $?x_7(x)$ into x.

```
by (resolve_tac [basic] 1);
  Level 7
   |- EX y. ALL x. P(y) --> P(x)
  No subgoals!
```

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

```
Goal "|- EX y. ALL x. P(y)-->P(x)";
  Level 0
   |- EX y. ALL x. P(y) --> P(x)
   1.  |- EX y. ALL x. P(y) --> P(x)
by (best_tac LK_dup_pack 1);
  Level 1
   |- EX y. ALL x. P(y) --> P(x)
  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 set — a set consisting of those sets that are not members of themselves:

$$\vdash \neg(\exists x\,.\,\forall y\,.\,y \in x \leftrightarrow y \notin y)$$

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 `Fast_tac`, has a short manual proof. See the directory `Sequents/LK` for many more examples.

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

```
Goal "|- ~ (EX x. ALL y. F(y,x) <-> ~F(y,y))";
  Level 0
   |- ~ (EX x. ALL y. F(y,x) <-> ~ F(y,y))
   1.  |- ~ (EX x. ALL y. F(y,x) <-> ~ F(y,y))
by (resolve_tac [notR] 1);
  Level 1
   |- ~ (EX x. ALL y. F(y,x) <-> ~ F(y,y))
   1. EX x. ALL y. F(y,x) <-> ~ F(y,y) |-
```

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

```
by (resolve_tac [exL] 1);
  Level 2
   |- ~ (EX x. ALL y. F(y,x) <-> ~ F(y,y))
   1. !!x. ALL y. F(y,x) <-> ~ F(y,y) |-
by (resolve_tac [allL_thin] 1);
  Level 3
   |- ~ (EX x. ALL y. F(y,x) <-> ~ F(y,y))
   1. !!x. F(?x2(x),x) <-> ~ F(?x2(x),?x2(x)) |-
```

The rule **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
   |- ~ (EX x. ALL y. F(y,x) <-> ~ F(y,y))
   1. !!x.  |- F(?x2(x),x), ~ F(?x2(x),?x2(x))
   2. !!x. ~ F(?x2(x),?x2(x)), F(?x2(x),x) |-
```

We must instantiate $?x_2$, 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
   |- ~ (EX x. ALL y. F(y,x) <-> ~ F(y,y))
   1. !!x.  |- F(?x2(x),x), ~ F(?x2(x),?x2(x))
   2. !!x. F(?x2(x),x) |- F(?x2(x),?x2(x))
by (resolve_tac [basic] 2);
  Level 6
   |- ~ (EX x. ALL y. F(y,x) <-> ~ F(y,y))
   1. !!x.  |- F(x,x), ~ F(x,x)
```

Thanks to the instantiation of $?x_2$, subgoal 1 is obviously true.

```
by (resolve_tac [notR] 1);
  Level 7
   |- ~ (EX x. ALL y. F(y,x) <-> ~ F(y,y))
   1. !!x. F(x,x) |- F(x,x)
by (resolve_tac [basic] 1);
  Level 8
   |- ~ (EX x. ALL y. F(y,x) <-> ~ 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 $\lambda x\,.\,?f(?g(x))$ 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:

$$\frac{\Gamma, P, Q, \Delta \vdash \Theta}{\Gamma, P \land Q, \Delta \vdash \Theta}\ (\land\text{-left})$$

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 `allL_thin` and `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.

`prop_pack` contains the propositional rules, namely those for $\land$, $\lor$, $\neg$, $\to$ and $\leftrightarrow$, along with the rules `basic` and `refl`. These are all safe.

`LK_pack` extends `prop_pack` with the safe rules `allR` and `exL` and the unsafe rules `allL_thin` and `exR_thin`. Search using this is incomplete since quantified formulae are used at most once.

`LK_dup_pack` extends `prop_pack` with the safe rules `allR` and `exL` and the unsafe rules `allL` and `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.

pack `add_safes` *rules* adds some safe *rules* to the pack *pack*.

pack `add_unsafes` *rules* adds some unsafe *rules* to the pack *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

$$\begin{array}{l} P(0), P(1), P(2) \vdash P(?a) \\ P(0), P(2), P(3) \vdash P(?a) \\ P(1), P(3), P(2) \vdash P(?a) \end{array}$$

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

3.9.1 Method A

```
reresolve_tac   : thm list -> int -> tactic
repeat_goal_tac : pack -> int -> tactic
pc_tac          : pack -> int -> tactic
```

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.

`reresolve_tac` *thms i* repeatedly applies the *thms* to subgoal *i* and the resulting subgoals.

`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.

`pc_tac` *pack i* applies `repeat_goal_tac` using depth-first search to solve subgoal *i*.

3.9.2 Method B

```
safe_tac : pack -> int -> tactic
step_tac : pack -> int -> tactic
fast_tac : pack -> int -> tactic
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.

`safe_goal_tac` *pack i* applies the safe rules in the pack to a goal and the resulting subgoals. It ignores the unsafe rules.

`step_tac` *pack i* either applies safe rules (using `safe_goal_tac`) or applies one unsafe rule.

`fast_tac` *pack i* applies `step_tac` using depth-first search to solve subgoal *i*. Despite its name, it is frequently slower than `pc_tac`.

`best_tac` *pack i* applies `step_tac` using best-first search to solve subgoal *i*. It is particularly useful for quantifier duplication (using `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 |- 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) |- 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 `|- 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 `seq`. A `seq` is created either by the empty space, a `seqobj` or a `seqobj` followed by a `seq`, with a comma between them. A `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 `[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 *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 *abstract* function `SeqO'` with type `[o,seq']=>seq'`, and translating a sequence such as `A, B, C` into

```
%s. SeqO'(A, SeqO'(B, SeqO'(C, s)))
```

This sequence can therefore be seen as a constructor for further sequences. The constructor `SeqO'` 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 `A, $B, C`, we note that `$B` already represents a sequence, so we can use `B` itself to refer to the function, and therefore the sequence can be mapped to the internal form: `%s. SeqO'(A, B(SeqO'(C, s)))`.

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

4.3 Object logics

Recall that object logics are defined by mapping elements of particular types to the Isabelle type `prop`, usually with a function called `Trueprop`. So, an object logic proposition `P` is matched to the Isabelle proposition `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 `Trueprop` automatically when an object of type `prop` is expected. This mechanism can be observed in most of the object logics which are direct descendants of `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 `Sequents/LK.thy`:

```
Trueprop       :: two_seqi
"@Trueprop"    :: two_seqe   ("((_)/ |- (_))" [6,6] 5)
```

where the types used are defined in `Sequents.thy` as abbreviations:

```
two_seqi = [seq'=>seq', seq'=>seq'] => prop
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 ML. Fortunately the theory developer need not delve in this level; `Sequents.thy` provides the necessary facilities. All the theory developer needs to add in the ML section is a specification of the two translation functions:

```
ML
val parse_translation = [("@Trueprop",Sequents.two_seq_tr "Trueprop")];
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 `Sequents.thy`

Theory `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`.

Chapter 5

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

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

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 $\bigwedge$ and $\Longrightarrow$:

$$\begin{gathered}\textstyle\bigwedge x_1 \ldots x_n . [\![x_1 \in A_1; x_2 \in A_2(x_1); \cdots\ x_n \in A_n(x_1,\ldots,x_{n-1})]\!] \Longrightarrow \\ a(x_1,\ldots,x_n) \in A(x_1,\ldots,x_n)\end{gathered}$$

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

$$\bigwedge x\,.\, x \in A \Longrightarrow B(x)\ \text{type}$$

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.

name	*meta-type*	*description*
`Type`	$t \to prop$	judgement form
`Eqtype`	$[t, t] \to prop$	judgement form
`Elem`	$[i, t] \to prop$	judgement form
`Eqelem`	$[i, i, t] \to prop$	judgement form
`Reduce`	$[i, i] \to prop$	extra judgement form
`N`	t	natural numbers type
`0`	i	constructor
`succ`	$i \to i$	constructor
`rec`	$[i, i, [i, i] \to i] \to i$	eliminator
`Prod`	$[t, i \to t] \to t$	general product type
`lambda`	$(i \to i) \to i$	constructor
`Sum`	$[t, i \to t] \to t$	general sum type
`pair`	$[i, i] \to i$	constructor
`split`	$[i, [i, i] \to i] \to i$	eliminator
`fst snd`	$i \to i$	projections
`inl inr`	$i \to i$	constructors for +
`when`	$[i, i \to i, i \to i] \to i$	eliminator for +
`Eq`	$[t, i, i] \to t$	equality type
`eq`	i	constructor
`F`	t	empty type
`contr`	$i \to i$	eliminator
`T`	t	singleton type
`tt`	i	constructor

Figure 5.1: The constants of CTT

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, `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 `Sum(`A`,`B`)` and `Prod(`A`,`B`)` we may write `SUM` x`:`A`.` $B[x]$ and `PROD` x`:`A`.` $B[x]$. For example, we may write

```
SUM y:B. PROD x:A. C(x,y)    for    Sum(B, %y. Prod(A, %x. C(x,y)))
```

The special cases as A`*`B and A`-->`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, `*` and `-->` merely define abbreviations; there are no constants `op *` and `op -->`.

symbol	*name*	*meta-type*	*priority*	*description*
lam	lambda	$(i \Rightarrow o) \Rightarrow i$	10	λ-abstraction

Binders

symbol	*meta-type*	*priority*	*description*
‘	$[i, i] \to i$	Left 55	function application
+	$[t, t] \to t$	Right 30	sum of two types

Infixes

external	*internal*	*standard notation*
PROD $x{:}A$. $B[x]$	Prod(A, $\lambda x . B[x]$)	product $\prod_{x \in A} B[x]$
SUM $x{:}A$. $B[x]$	Sum(A, $\lambda x . B[x]$)	sum $\sum_{x \in A} B[x]$
A --> B	Prod(A, $\lambda x . B$)	function space $A \to B$
A * B	Sum(A, $\lambda x . B$)	binary product $A \times B$

Translations

$$
\begin{array}{rcl}
\textit{prop} & = & \textit{type}\ \texttt{type} \\
 & | & \textit{type}\ \texttt{=}\ \textit{type} \\
 & | & \textit{term}\ \texttt{:}\ \textit{type} \\
 & | & \textit{term}\ \texttt{=}\ \textit{term}\ \texttt{:}\ \textit{type} \\
 & & \\
\textit{type} & = & \text{expression of type } t \\
 & | & \texttt{PROD}\ \textit{id}\ \texttt{:}\ \textit{type}\ \texttt{.}\ \textit{type} \\
 & | & \texttt{SUM}\ \ \textit{id}\ \texttt{:}\ \textit{type}\ \texttt{.}\ \textit{type} \\
 & & \\
\textit{term} & = & \text{expression of type } i \\
 & | & \texttt{lam}\ \textit{id}\ \textit{id}^{*}\ \texttt{.}\ \textit{term} \\
 & | & \texttt{<}\ \textit{term}\ \texttt{,}\ \textit{term}\ \texttt{>}
\end{array}
$$

Grammar

Figure 5.2: Syntax of CTT

```
refl_type         A type ==> A = A
refl_elem         a : A ==> a = a : A

sym_type          A = B ==> B = A
sym_elem          a = b : A ==> b = a : A

trans_type        [| A = B;  B = C |] ==> A = C
trans_elem        [| a = b : A;  b = c : A |] ==> a = c : A

equal_types       [| a : A;  A = B |] ==> a : B
equal_typesL      [| a = b : A;  A = B |] ==> a = b : B

subst_type        [| a : A;  !!z. z:A ==> B(z) type |] ==> B(a) type
subst_typeL       [| a = c : A;  !!z. z:A ==> B(z) = D(z)
                  |] ==> B(a) = D(c)

subst_elem        [| a : A;  !!z. z:A ==> b(z):B(z) |] ==> b(a):B(a)
subst_elemL       [| a = c : A;  !!z. z:A ==> b(z) = d(z) : B(z)
                  |] ==> b(a) = d(c) : B(a)

refl_red          Reduce(a,a)
red_if_equal      a = b : A ==> Reduce(a,b)
trans_red         [| a = b : A;  Reduce(b,c) |] ==> a = c : A
```

Figure 5.3: General equality rules

```
NF        N type

NI0       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)

NC0       [| 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 $\prod_{x\in A} B[x]$

```
SumF      [| A type;  !!x. x:A ==> B(x) type |] ==> SUM x:A. B(x) type
SumFL     [| A = C;  !!x. x:A ==> B(x) = D(x)
          |] ==> SUM x:A. B(x) = SUM x:C. D(x)

SumI      [| a : A;  b : B(a) |] ==> <a,b> : SUM x:A. B(x)
SumIL     [| a=c:A;  b=d:B(a) |] ==> <a,b> = <c,d> : SUM x:A. B(x)

SumE      [| p: SUM x:A. B(x);
             !!x y. [| x:A; y:B(x) |] ==> c(x,y): C(<x,y>)
          |] ==> split(p, %x y. c(x,y)) : C(p)

SumEL     [| p=q : SUM x:A. B(x);
             !!x y. [| x:A; y:B(x) |] ==> c(x,y)=d(x,y): C(<x,y>)
          |] ==> split(p, %x y. c(x,y)) = split(q, %x y. d(x,y)) : C(p)

SumC      [| a: A;  b: B(a);
             !!x y. [| x:A; y:B(x) |] ==> c(x,y): C(<x,y>)
          |] ==> split(<a,b>, %x y. c(x,y)) = c(a,b) : C(<a,b>)

fst_def   fst(a) == split(a, %x y. x)
snd_def   snd(a) == 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 Reduce does not belong to Type Theory proper; it has been added to implement rewriting. The judgement 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 refl_red does not verify that a belongs to A.

The Reduce rules do not give rise to new theorems about the standard judgements. The only rule with Reduce in a premise is 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 zero_ne_succ, which asserts $0 \neq n + 1$. This is the fourth Peano

```
PlusF       [| A type;  B type |] ==> A+B type
PlusFL      [| A = C;  B = D |] ==> A+B = C+D

PlusI_inl   [| a : A;  B type |] ==> inl(a) : A+B
PlusI_inlL  [| a = c : A;  B type |] ==> inl(a) = inl(c) : A+B

PlusI_inr   [| A type;  b : B |] ==> inr(b) : A+B
PlusI_inrL  [| A type;  b = d : B |] ==> inr(b) = inr(d) : A+B

PlusE     [| p: A+B;
             !!x. x:A ==> c(x): C(inl(x));
             !!y. y:B ==> d(y): C(inr(y))
          |] ==> when(p, %x. c(x), %y. d(y)) : C(p)

PlusEL    [| p = q : A+B;
             !!x. x: A ==> c(x) = e(x) : C(inl(x));
             !!y. y: B ==> d(y) = f(y) : C(inr(y))
          |] ==> when(p, %x. c(x), %y. d(y)) =
                 when(q, %x. e(x), %y. f(y)) : C(p)

PlusC_inl [| a: A;
             !!x. x:A ==> c(x): C(inl(x));
             !!y. y:B ==> d(y): C(inr(y))
          |] ==> when(inl(a), %x. c(x), %y. d(y)) = c(a) : C(inl(a))

PlusC_inr [| b: B;
             !!x. x:A ==> c(x): C(inl(x));
             !!y. y:B ==> d(y): C(inr(y))
          |] ==> when(inr(b), %x. c(x), %y. d(y)) = d(b) : C(inr(b))
```

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

```
FF        F type
FE        [| p: F;  C type |] ==> contr(p) : C
FEL       [| p = q : F;  C type |] ==> contr(p) = contr(q) : C

TF        T type
TI        tt : T
TE        [| p : T;  c : C(tt) |] ==> c : C(p)
TEL       [| p = q : T;  c = d : C(tt) |] ==> c = d : C(p)
TC        p : T ==> p = tt : T)
```

Figure 5.8: Rules for types F and T

```
EqF      [| A type;  a : A;  b : A |] ==> Eq(A,a,b) type
EqFL     [| A=B;  a=c: A;  b=d : A |] ==> Eq(A,a,b) = Eq(B,c,d)
EqI      a = b : A ==> eq : Eq(A,a,b)
EqE      p : Eq(A,a,b) ==> a = b : A
EqC      p : Eq(A,a,b) ==> p = eq : Eq(A,a,b)
```

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

```
replace_type    [| B = A;  a : A |] ==> a : B
subst_eqtyparg  [| a=c : A;  !!z. z:A ==> B(z) type |] ==> B(a)=B(c)

subst_prodE     [| p: Prod(A,B);  a: A;  !!z. z: B(a) ==> c(z): C(z)
                |] ==> c(p`a): C(p`a)

SumIL2    [| c=a : A;  d=b : B(a) |] ==> <c,d> = <a,b> : Sum(A,B)

SumE_fst  p : Sum(A,B) ==> fst(p) : A

SumE_snd  [| p: Sum(A,B);  A type;  !!x. x:A ==> B(x) type
          |] ==> snd(p) : B(fst(p))
```

Figure 5.10: Derived rules for CTT

axiom and cannot be derived without universes [9, page 91].

The constant `rec` constructs proof terms when mathematical induction, rule `NE`, is applied. It can also express primitive recursion. Since `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 λ-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 `fst` and `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 `inl` and `inr` and case analysis operator `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 eq is a canonical element of the type $Eq(A, a, b)$, 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 subst_prodE is derived from prodE, and is easier to use in backwards proof. The rules SumE_fst and SumE_snd express the typing of fst and snd; together, they are roughly equivalent to 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 η-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 thm list — of related rules.

form_rls contains formation rules for the types N, Π, Σ, $+$, Eq, F, and T.

formL_rls contains long formation rules for Π, Σ, $+$, and Eq. (For other types use refl_type.)

intr_rls contains introduction rules for the types N, Π, Σ, $+$, and T.

intrL_rls contains long introduction rules for N, Π, Σ, and $+$. (For T use refl_elem.)

elim_rls contains elimination rules for the types N, Π, Σ, $+$, and F. The rules for Eq and T are omitted because they involve no eliminator.

elimL_rls contains long elimination rules for N, Π, Σ, $+$, and F.

comp_rls contains computation rules for the types N, Π, Σ, and $+$. Those for Eq and T involve no eliminator.

basic_defs contains the definitions of fst and snd.

5.4 Tactics for subgoal reordering

```
test_assume_tac : int -> tactic
typechk_tac     : thm list -> tactic
equal_tac       : thm list -> tactic
intr_tac        : thm list -> tactic
```

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 `filt_resolve_tac` (see the *Reference Manual*) fails for goals that are too flexible; so does the CTT tactic `test_assume_tac`. Used with the tactical `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.

`test_assume_tac` i uses `assume_tac` to solve the subgoal by assumption, but only if subgoal i has the form $a \in A$ and the head of a is not an unknown. Otherwise, it fails.

`typechk_tac` *thms* uses *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.

`equal_tac` *thms* uses *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.

`intr_tac` *thms* uses *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

```
rew_tac     : thm list -> tactic
hyp_rew_tac : thm list -> tactic
```

Object-level simplification is accomplished through proof, using the `CTT` equality rules and the built-in rewriting functor `TSimpFun`.[2] The rewrites

[2] This should not be confused with Isabelle's main simplifier; `TSimpFun` is only useful for CTT and similar logics with type inference rules. At present it is undocumented.

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
safestep_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

$$a \bmod b + (a/b) \times b = a.$$

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 $\lambda v\,.\,\mathtt{rec}(v, 0, \lambda x\, y\,.\, x)$.

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

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

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.

symbol	*meta-type*	*priority*	*description*
#*	$[i,i] \Rightarrow i$	Left 70	multiplication
div	$[i,i] \Rightarrow i$	Left 70	division
mod	$[i,i] \Rightarrow i$	Left 70	modulus
#+	$[i,i] \Rightarrow i$	Left 65	addition
-	$[i,i] \Rightarrow i$	Left 65	subtraction
|-|	$[i,i] \Rightarrow i$	Left 65	absolute difference

```
add_def           a#+b  == rec(a, b, %u v. succ(v))
diff_def          a-b   == rec(b, a, %u v. rec(v, 0, %x y. x))
absdiff_def       a|-|b == (a-b) #+ (b-a)
mult_def          a#*b  == rec(a, 0, %u v. b #+ v)

mod_def           a mod b ==
                  rec(a, 0, %u v. rec(succ(v) |-| b, 0, %x y. succ(v)))

div_def           a div b ==
                  rec(a, 0, %u v. rec(succ(u) mod b, succ(v), %x y. v))

add_typing        [| a:N;  b:N |] ==> a #+ b : N
addC0             b:N ==> 0 #+ b = b : N
addC_succ         [| a:N;  b:N |] ==> succ(a) #+ b = succ(a #+ b) : N

add_assoc         [| a:N;  b:N;  c:N |] ==>
                  (a #+ b) #+ c = a #+ (b #+ c) : N

add_commute       [| a:N;  b:N |] ==> a #+ b = b #+ a : N

mult_typing       [| a:N;  b:N |] ==> a #* b : N
multC0            b:N ==> 0 #* b = 0 : N
multC_succ        [| a:N;  b:N |] ==> succ(a) #* b = b #+ (a#*b) : N
mult_commute      [| a:N;  b:N |] ==> a #* b = b #* a : N

add_mult_dist     [| a:N;  b:N;  c:N |] ==>
                  (a #+ b) #* c = (a #* c) #+ (b #* c) : N

mult_assoc        [| a:N;  b:N;  c:N |] ==>
                  (a #* b) #* c = a #* (b #* c) : N

diff_typing       [| a:N;  b:N |] ==> a - b : N
diffC0            a:N ==> a - 0 = a : N
diff_0_eq_0       b:N ==> 0 - b = 0 : N
diff_succ_succ    [| a:N;  b:N |] ==> succ(a) - succ(b) = a - b : N
diff_self_eq_0    a:N ==> a - a = 0 : N
add_inverse_diff  [| a:N;  b:N;  b-a=0 : N |] ==> 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 pc_tac.

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

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

5.9 Example: type inference

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

```
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 λ-abstraction (not to be confused with a meta-level abstraction), we apply the rule ProdI, for Π-introduction. This instantiates $?A$ 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 $?A_1$ 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 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 $?A_1$ 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 [NIO] 2);
  Level 3
  lam n. rec(n,0,%x y. x) : N --> N
   1. N type
   2. !!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 `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:

```
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:

$$\frac{\exists x \in A \,.\, P(x) \lor Q(x)}{(\exists x \in A \,.\, P(x)) \lor (\exists x \in A \,.\, Q(x))}$$

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

$$\frac{A \times (B + C)}{(A \times B) + (A \times C)}$$

Generalizing this from $\times$ to Σ, and making the typing conditions explicit, yields the rule we must derive:

$$\frac{A\ \mathrm{type} \quad \begin{matrix}[x\in A]\\ \vdots \\ B(x)\ \mathrm{type}\end{matrix} \quad \begin{matrix}[x\in A]\\ \vdots \\ C(x)\ \mathrm{type}\end{matrix} \quad p \in \sum_{x\in A} B(x)+C(x)}{?a \in (\sum_{x\in A} B(x)) + (\sum_{x\in A} C(x))}$$

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

```
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))";
  Level 0
  ?a : (SUM x:A. B(x)) + (SUM x:A. C(x))
   1. ?a : (SUM x:A. B(x)) + (SUM x:A. C(x))
  val prems = ["A type  [A type]",
               "?x : A ==> B(?x) type  [!!x. x : A ==> B(x) type]",
               "?x : A ==> C(?x) type  [!!x. x : A ==> C(x) type]",
               "p : SUM x:A. B(x) + C(x)  [p : SUM x:A. B(x) + C(x)]"]
              : thm list
```

The last premise involves the sum type Σ. 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

```
cut_facts_tac prems 1;
```

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

```
by (resolve_tac (prems RL [SumE]) 1);
  Level 1
  split(p,?c1) : (SUM x:A. B(x)) + (SUM x:A. C(x))
   1. !!x y.
         [| x : A; y : B(x) + C(x) |] ==>
         ?c1(x,y) : (SUM x:A. B(x)) + (SUM x:A. C(x))
```

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
  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) |] ==>
         ?c2(x,y,xa) : (SUM x:A. B(x)) + (SUM x:A. C(x))
   2. !!x y ya.
         [| x : A; ya : C(x) |] ==>
         ?d2(x,y,ya) : (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 $?c_2(x, y, xa)$ and $?d_2(x, y, xa)$. We attack subgoal 1 by a +-introduction rule; since the goal assumes $xa \in B(x)$, we take the left injection (`inl`).

```
by (resolve_tac [PlusI_inl] 1);
  Level 3
  split(p,%x y. when(y,%xa. 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) |] ==> ?a3(x,y,xa) : SUM x:A. B(x)
   2. !!x y xa. [| x : A; xa : B(x) |] ==> SUM x:A. C(x) type
   3. !!x y ya.
         [| x : A; ya : C(x) |] ==>
         ?d2(x,y,ya) : (SUM x:A. B(x)) + (SUM x:A. C(x))
```

A new subgoal 2 has appeared, to verify that $\sum_{x \in A} C(x)$ is a type. Continuing to work on subgoal 1, we apply the Σ-introduction rule. This instantiates the term $?a_3(x, y, xa)$; the main goal now contains an ordered pair, whose components are two new unknowns.

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

The two new subgoals both hold by assumption. Observe how the unknowns $?a_4$ and $?b_4$ are instantiated throughout the proof state.

```
by (assume_tac 1);
  Level 5
  split(p,%x y. when(y,%xa. inl(<x,?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) |] ==> ?b4(x,y,xa) : B(x)
     2. !!x y xa. [| x : A; xa : B(x) |] ==> SUM x:A. C(x) type
     3. !!x y ya.
           [| x : A; ya : C(x) |] ==>
           ?d2(x,y,ya) : (SUM x:A. B(x)) + (SUM x:A. C(x))
by (assume_tac 1);
  Level 6
  split(p,%x y. when(y,%xa. inl(<x,xa>),?d2(x,y)))
  : (SUM x:A. B(x)) + (SUM x:A. C(x))
   1. !!x y xa. [| x : A; xa : B(x) |] ==> SUM x:A. C(x) type
   2. !!x y ya.
         [| x : A; ya : C(x) |] ==>
         ?d2(x,y,ya) : (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)))
  : (SUM x:A. B(x)) + (SUM x:A. C(x))
   1. !!x y ya.
         [| x : A; ya : C(x) |] ==>
         ?d2(x,y,ya) : (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 Σ-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

$$(A \times B \to C) \to (A \to (B \to C)).$$

Let us generalize this to the dependent types Σ and Π. 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 $C(\langle x, y\rangle)$.

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 `PROD f` to ensure that the parameter corresponding to the functional's argument is really called f; Isabelle echoes the type using `-->` 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 : PROD f: (PROD z : (SUM x:A . B(x)) . C(z)).        \
\                       (PROD x:A . PROD y:B(x) . C(<x,y>))";
  Level 0
  ?a : (PROD z:SUM x:A. B(x). C(z)) -->
       (PROD x:A. PROD y:B(x). C(<x,y>))
   1. ?a : (PROD z:SUM x:A. B(x). C(z)) -->
           (PROD x:A. 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
```

This is a chance to demonstrate `intr_tac`. Here, the tactic repeatedly applies Π-introduction and proves the rather tiresome typing conditions.

Note that $?a$ becomes instantiated to three nested λ-abstractions. It would be easier to read if the bound variable names agreed with the parameters in the subgoal. Isabelle attempts to give parameters the same names as corresponding bound variables in the goal, but this does not always work. In any event, the goal is logically correct.

```
by (intr_tac prems);
  Level 1
  lam x xa xb. ?b7(x,xa,xb)
  : (PROD z:SUM x:A. B(x). C(z)) --> (PROD x:A. PROD y:B(x). C(<x,y>))
   1. !!f x y.
         [| f : PROD z:SUM x:A. B(x). C(z); x : A; y : B(x) |] ==>
         ?b7(f,x,y) : C(<x,y>)
```

Using Π-elimination, we solve subgoal 1 by applying the function f.

```
by (eresolve_tac [ProdE] 1);
  Level 2
  lam x xa xb. x ` <xa,xb>
  : (PROD z:SUM x:A. B(x). C(z)) --> (PROD x:A. PROD y:B(x). C(<x,y>))
   1. !!f x y. [| x : A; y : B(x) |] ==> <x,y> : SUM x:A. B(x)
```

Finally, we verify that the argument's type is suitable for the function application. This is straightforward using introduction rules.

```
by (intr_tac prems);
  Level 3
  lam x xa xb. x ` <xa,xb>
  : (PROD z:SUM x:A. B(x). C(z)) --> (PROD x:A. PROD y:B(x). C(<x,y>))
  No subgoals!
```

Calling `pc_tac` would have proved this theorem in one step; it can also prove an example by Martin-Löf, related to $\vee$-elimination [9, page 58].

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:

$$\begin{aligned} f &\equiv \texttt{fst} \circ h \\ g &\equiv \texttt{snd} \circ h \end{aligned}$$

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.

```
val prems = Goal
    "[| A type;  !!x. x:A ==> B(x) type;                    \
\       !!x y.[| x:A;  y:B(x) |] ==> C(x,y) type            \
\    |] ==> ?a : PROD h: (PROD x:A. SUM y:B(x). C(x,y)).    \
\                     (SUM f: (PROD x:A. B(x)). PROD x:A. C(x, f`x))";
  Level 0
  ?a : (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. ?a : (PROD x:A. SUM y:B(x). C(x,y)) -->
           (SUM f:PROD x:A. B(x). PROD x:A. C(x,f ` x))
  val prems = ["A type  [A type]",
               "?x : A ==> B(?x) type  [!!x. x : A ==> B(x) type]",
               "[| ?x : A; ?y : B(?x) |] ==> C(?x, ?y) type
                [!!x y. [| x : A; y : B(x) |] ==> C(x, y) type]"]
              : thm list
```

First, `intr_tac` applies introduction rules and performs routine type-checking. This instantiates $?a$ to a construction involving a λ-abstraction and an ordered pair. The pair's components are themselves λ-abstractions and there is a subgoal for each.

```
by (intr_tac prems);
  Level 1
  lam x. <lam xa. ?b7(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 |] ==>
         ?b7(h,x) : B(x)
   2. !!h x.
         [| h : PROD x:A. SUM y:B(x). C(x,y); x : A |] ==>
         ?b8(h,x) : C(x,(lam x. ?b7(h,x)) ` x)
```

Subgoal 1 asks to find the choice function itself, taking $x \in A$ to some $?b_7(h,x) \in B(x)$. Subgoal 2 asks, given $x \in A$, for a proof object $?b_8(h,x)$ 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.

```
by (eresolve_tac [ProdE RS SumE_fst] 1);
  Level 2
  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. x : A ==> x : A
   2. !!h x.
         [| h : PROD x:A. SUM y:B(x). C(x,y); x : A |] ==>
         ?b8(h,x) : C(x,(lam x. fst(h ` x)) ` x)
```

Above, we have composed `fst` with the function h. Unification has deduced that the function must be applied to $x \in A$. We have our choice function.

```
by (assume_tac 1);
  Level 3
  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,(lam x. fst(h ` x)) ` x)
```

Before we can compose `snd` with h, the arguments of C must be simplified. The derived rule `replace_type` lets us replace a type by any equivalent type, shown below as the schematic term $?A_{13}(h,x)$:

```
by (resolve_tac [replace_type] 1);
  Level 4
  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 |] ==>
       C(x,(lam x. fst(h ` x)) ` x) = ?A13(h,x)
 2. !!h x.
       [| h : PROD x:A. SUM y:B(x). C(x,y); x : A |] ==>
       ?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):

```
by (resolve_tac [subst_eqtyparg] 1);
  Level 5
  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 |] ==>
         (lam x. fst(h ` x)) ` x = ?c14(h,x) : ?A14(h,x)
   2. !!h x z.
         [| h : PROD x:A. SUM y:B(x). C(x,y); x : A;
            z : ?A14(h,x) |] ==>
         C(x,z) type
   3. !!h x.
         [| h : PROD x:A. SUM y:B(x). C(x,y); x : A |] ==>
         ?b8(h,x) : C(x,?c14(h,x))
```

Subgoal 1 requires simply β-contraction, which is the rule `ProdC`. The term $?c_{14}(h,x)$ in the last subgoal receives the contracted result.

```
by (resolve_tac [ProdC] 1);
  Level 6
  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 |] ==>
         x : ?A15(h,x)
   2. !!h x xa.
         [| h : PROD x:A. SUM y:B(x). C(x,y); x : A;
            xa : ?A15(h,x) |] ==>
         fst(h ` xa) : ?B15(h,x,xa)
   3. !!h x z.
         [| h : PROD x:A. SUM y:B(x). C(x,y); x : A;
            z : ?B15(h,x,x) |] ==>
         C(x,z) type
   4. !!h x.
         [| h : PROD x:A. SUM y:B(x). C(x,y); x : A |] ==>
         ?b8(h,x) : C(x,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

[1] Stefan Berghofer and Markus Wenzel. Inductive datatypes in HOL — lessons learned in Formal-Logic Engineering. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics: TPHOLs '99*, volume 1690 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[2] Martin D. Coen. *Interactive Program Derivation*. PhD thesis, University of Cambridge, November 1992. Computer Laboratory Technical Report 272.

[3] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.

[4] Amy Felty. A logic program for transforming sequent proofs to natural deduction proofs. In Peter Schroeder-Heister, editor, *Extensions of Logic Programming*, LNAI 475, pages 157–178. Springer, 1991.

[5] J. H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Harper & Row, 1986.

[6] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[7] G. P. Huet and B. Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[8] Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs: International Workshop TYPES '93*, LNCS 806, pages 213–237. Springer, published 1994.

[9] Per Martin-Löf. *Intuitionistic type theory*. Bibliopolis, 1984.

[10] Olaf Müller, Tobias Nipkow, David von Oheimb, and Oscar Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9:191–223, 1999.

[11] Bengt Nordström, Kent Petersson, and Jan Smith. *Programming in Martin-Löf's Type Theory. An Introduction.* Oxford University Press, 1990.

[12] Lawrence C. Paulson. *Isabelle's Logics: FOL and ZF.* http://isabelle.in.tum.de/doc/logics-ZF.pdf.

[13] Lawrence C. Paulson. *Logic and Computation: Interactive proof with Cambridge LCF.* Cambridge University Press, 1987.

[14] Lawrence C. Paulson. A formulation of the simple theory of types (for Isabelle). In P. Martin-Löf and G. Mints, editors, *COLOG-88: International Conference on Computer Logic*, LNCS 417, pages 246–274, Tallinn, Published 1990. Estonian Academy of Sciences, Springer.

[15] F. J. Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, 2:191–216, 1986. Errata, JAR 4 (1988), 235–236 and JAR 18 (1997), 135.

[16] Konrad Slind. Function definition in higher order logic. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics: TPHOLs '96*, volume 1125 of *Lecture Notes in Computer Science*, pages 381–397. Springer-Verlag, 1996.

[17] G. Takeuti. *Proof Theory.* North-Holland, 2nd edition, 1987.

[18] Simon Thompson. *Type Theory and Functional Programming.* Addison-Wesley, 1991.

Index