Isabelle's Object-Logics

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

With Contributions by Tobias Nipkow and Markus Wenzel[1]

24 November 1997

[1]Tobias Nipkow revised and extended the chapter on `HOL`. Markus Wenzel made numerous improvements. Philippe de Groote wrote the first version of the logic `LK` and contributed to `ZF`. Tobias Nipkow developed `HOL`, `LCF` and `Cube`. Philippe Noël and Martin Coen made many contributions to `ZF`. 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) and by ESPRIT project 6453: Types.

Contents

Chapter 1

Basic Concepts

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.

`FOL` is many-sorted first-order logic with natural deduction. It comes in both constructive and classical versions.

`ZF` is axiomatic set theory, using the Zermelo-Fraenkel axioms [45]. It is built upon classical `FOL`.

`CCL` is Martin Coen's Classical Computational Logic, which is the basis of a preliminary method for deriving programs from proofs [7]. 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 [32]. It is built upon classical `FOL`.

`HOL` is the higher-order logic of Church [6], which is also implemented by Gordon's HOL system [16]. This object-logic should not be confused with Isabelle's meta-logic, which is also a form of higher-order logic.

`HOLCF` is a version of LCF, defined as an extension of `HOL`.

`CTT` is a version of Martin-Löf's Constructive Type Theory [29], with extensional equality. Universes are not included.

`LK` is another version of first-order logic, a classical sequent calculus. Sequents have the form $A_1, \ldots, A_m \vdash B_1, \ldots, B_n$; rules are applied using associative matching.

`Modal` implements the modal logics T, $S4$, and $S43$. It is built upon `LK`.

`Cube` is Barendregt's λ-cube.

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

```
http://www4.informatik.tu-muenchen.de/~nipkow/isabelle/
```

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

You should not read this manual before reading *Introduction to Isabelle* and performing some Isabelle proofs. Consult the *Reference Manual* for more information on tactics, packages, etc.

1.1 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. `HOL`'s description operator $\varepsilon x \,.\, P\, x$ has type $(\alpha \Rightarrow \mathit{bool}) \Rightarrow \alpha$ and can bind only one variable, except when α is *bool*. `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.

1.2 Proof procedures

Most object-logics come with simple proof procedures. These are reasonably powerful for interactive use, though often simplistic and incomplete. You can do single-step proofs using `resolve_tac` and `assume_tac`, referring to the inference rules of the logic by ML identifiers.

For theorem proving, rules can 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 rules of the classical sequent calculus LK are safe. Universal elimination is unsafe if the formula $\forall x \,.\, P(x)$ is deleted after use. Other unsafe rules include the following:

$$\frac{P}{P \lor Q}\ (\lor I1) \qquad \frac{P \to Q \quad P}{Q}\ (\to E) \qquad \frac{P[t/x]}{\exists x \,.\, P}\ (\exists I)$$

Proof procedures use safe rules whenever possible, delaying the application of unsafe rules. Those safe rules are preferred that generate the fewest subgoals. Safe rules are (by definition) deterministic, while the unsafe rules require search. The design of a suitable set of rules can be as important as the strategy for applying them.

Many of the proof procedures use backtracking. Typically they attempt to solve subgoal i by repeatedly applying a certain tactic to it. This tactic, which is known as a **step tactic**, resolves a selection of rules with subgoal i. This may replace one subgoal by many; the search persists until there are fewer subgoals in total than at the start. Backtracking happens when the search reaches a dead end: when the step tactic fails. Alternative outcomes are then searched by a depth-first or best-first strategy.

Chapter 2

First-Order Logic

Isabelle implements Gentzen's natural deduction systems NJ and NK. Intuitionistic first-order logic is defined first, as theory `IFOL`. Classical logic, theory `FOL`, is obtained by adding the double negation rule. Basic proof procedures are provided. The intuitionistic prover works with derived rules to simplify implications in the assumptions. Classical `FOL` employs Isabelle's classical reasoner, which simulates a sequent calculus.

2.1 Syntax and rules of inference

The logic is many-sorted, using Isabelle's type classes. The class of first-order terms is called `term` and is a subclass of `logic`. No types of individuals are provided, but extensions can define types such as `nat::term` and type constructors such as `list::(term)term` (see the examples directory, `FOL/ex`). 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`. Figure 2.1 gives the syntax. Note that a`~=`b is translated to $\neg(a = b)$.

Figure 2.2 shows the inference rules with their ML names. Negation is defined in the usual way for intuitionistic logic; $\neg P$ abbreviates $P \to \bot$. The biconditional ($\leftrightarrow$) is defined through $\land$ and $\to$; introduction and elimination rules are derived for it.

The unique existence quantifier, $\exists! x \,.\, P(x)$, 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 $P(x, y)$.

Some intuitionistic derived rules are shown in Fig. 2.3, again with their ML names. These include rules for the defined symbols $\neg$, $\leftrightarrow$ and $\exists!$. Natural deduction typically involves a combination of forward and backward reasoning, particularly with the destruction rules $(\land E)$, $(\to E)$, and $(\forall E)$. Isabelle's backward style handles these rules badly, so sequent-style rules are derived to eliminate conjunctions, implications, and universal quantifiers. Used with elim-resolution, `allE` eliminates a universal quantifier while `all_dupE` re-inserts the quantified formula for later use. The rules `conj_impE`, etc., support the intuitionistic proof procedure (see §2.3).

See the files FOL/IFOL.thy, FOL/IFOL.ML and FOL/intprover.ML for complete listings of the rules and derived rules.

2.2 Generic packages

FOL instantiates most of Isabelle's generic packages.

- It instantiates the simplifier. Both equality (=) and the biconditional ($\leftrightarrow$) may be used for rewriting. Tactics such as Asm_simp_tac and Full_simp_tac use the default simpset (!simpset), which works for most purposes. Named simplification sets include IFOL_ss, for intuitionistic first-order logic, and FOL_ss, for classical logic. See the file FOL/simpdata.ML for a complete listing of the simplification rules.
- It instantiates the classical reasoner. See §2.4 for details.
- FOL provides the tactic hyp_subst_tac, which substitutes for an equality throughout a subgoal and its hypotheses. This tactic uses FOL's general substitution rule.

! 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].

2.3 Intuitionistic proof procedures

Implication elimination (the rules mp and impE) pose difficulties for automated proof. In intuitionistic logic, the assumption $P \to Q$ cannot be treated like $\neg P \lor Q$. Given $P \to Q$, we may use Q provided we can prove P; the proof of P may require repeated use of $P \to Q$. If the proof of P fails then the whole branch of the proof must be abandoned. Thus intuitionistic propositional logic requires backtracking.

For an elementary example, consider the intuitionistic proof of Q from $P \to Q$ and $(P \to Q) \to P$. The implication $P \to Q$ is needed twice:

$$\dfrac{P \to Q \qquad \dfrac{(P \to Q) \to P \quad P \to Q}{P}\,(\to E)}{Q}\,(\to E)$$

The theorem prover for intuitionistic logic does not use impE. Instead, it simplifies implications using derived rules (Fig. 2.3). It reduces the antecedents of

name	*meta-type*	*description*
`Trueprop`	$o \Rightarrow prop$	coercion to *prop*
`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 ($\exists$)
`EX!`	`Ex1`	$(\alpha \Rightarrow o) \Rightarrow o$	10	unique existence ($\exists!$)

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

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

Grammar

Figure 2.1: Syntax of `FOL`

```
refl        a=a
subst       [| a=b;  P(a) |] ==> P(b)
```

Equality rules

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

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

impI        (P ==> Q) ==> P-->Q
mp          [| P-->Q;  P |] ==> Q

FalseE      False ==> P
```

Propositional rules

```
allI        (!!x. P(x))  ==> (ALL x.P(x))
spec        (ALL x.P(x)) ==> P(x)

exI         P(x) ==> (EX x.P(x))
exE         [| EX x.P(x);  !!x. P(x) ==> R |] ==> R
```

Quantifier rules

```
True_def    True        == False-->False
not_def     ~P          == P-->False
iff_def     P<->Q       == (P-->Q) & (Q-->P)
ex1_def     EX! x. P(x) == EX x. P(x) & (ALL y. P(y) --> y=x)
```

Definitions

Figure 2.2: Rules of intuitionistic logic

```
sym       a=b ==> b=a
trans     [| a=b;  b=c |] ==> a=c
ssubst    [| b=a;  P(a) |] ==> P(b)
```

Derived equality rules

```
TrueI     True

notI      (P ==> False) ==> ~P
notE      [| ~P;  P |] ==> R

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

ex1I      [| P(a);  !!x. P(x) ==> x=a |]  ==>  EX! x. P(x)
ex1E      [| EX! x.P(x);  !!x.[| P(x);  ALL y. P(y) --> y=x |] ==> R
          |] ==> R
```

Derived rules for $\top$, $\neg$, $\leftrightarrow$ and $\exists!$

```
conjE     [| P&Q;  [| P; Q |] ==> R |] ==> R
impE      [| P-->Q;  P;  Q ==> R |] ==> R
allE      [| ALL x.P(x);  P(x) ==> R |] ==> R
all_dupE  [| ALL x.P(x);  [| P(x); ALL x.P(x) |] ==> R |] ==> R
```

Sequent-style elimination rules

```
conj_impE [| (P&Q)-->S;  P-->(Q-->S) ==> R |] ==> R
disj_impE [| (P|Q)-->S;  [| P-->S; Q-->S |] ==> R |] ==> R
imp_impE  [| (P-->Q)-->S;  [| P; Q-->S |] ==> Q;  S ==> R |] ==> R
not_impE  [| ~P --> S;  P ==> False;  S ==> R |] ==> R
iff_impE  [| (P<->Q)-->S; [| P; Q-->S |] ==> Q; [| Q; P-->S |] ==> P;
             S ==> R |] ==> R
all_impE  [| (ALL x.P(x))-->S;  !!x.P(x);  S ==> R |] ==> R
ex_impE   [| (EX x.P(x))-->S;  P(a)-->S ==> R |] ==> R
```

Intuitionistic simplification of implication

Figure 2.3: Derived rules for intuitionistic logic

implications to atoms and then uses Modus Ponens: from $P \to Q$ and P deduce Q. The rules `conj_impE` and `disj_impE` are straightforward: $(P \land Q) \to S$ is equivalent to $P \to (Q \to S)$, and $(P \lor Q) \to S$ is equivalent to the conjunction of $P \to S$ and $Q \to S$. The other ... `_impE` rules are unsafe; the method requires backtracking. All the rules are derived in the same simple manner.

Dyckhoff has independently discovered similar rules, and (more importantly) has demonstrated their completeness for propositional logic [12]. However, the tactics given below are not complete for first-order logic because they discard universally quantified assumptions after a single use.

```
mp_tac              : int -> tactic
eq_mp_tac           : int -> tactic
IntPr.safe_step_tac : int -> tactic
IntPr.safe_tac      :        tactic
IntPr.inst_step_tac : int -> tactic
IntPr.step_tac      : int -> tactic
IntPr.fast_tac      : int -> tactic
IntPr.best_tac      : int -> tactic
```

Most of these belong to the structure `IntPr` and resemble the tactics of Isabelle's classical reasoner.

`mp_tac` i attempts to use `notE` or `impE` within the assumptions in subgoal i. For each assumption of the form $\neg P$ or $P \to Q$, it searches for another assumption unifiable with P. By contradiction with $\neg P$ it can solve the subgoal completely; by Modus Ponens it can replace the assumption $P \to Q$ by Q. The tactic can produce multiple outcomes, enumerating all suitable pairs of assumptions.

`eq_mp_tac` i is like `mp_tac` i, but may not instantiate unknowns — thus, it is safe.

`IntPr.safe_step_tac` i performs a safe step on subgoal i. This may include proof by assumption or Modus Ponens (taking care not to instantiate unknowns), or `hyp_subst_tac`.

`IntPr.safe_tac` repeatedly performs safe steps on all subgoals. It is deterministic, with at most one outcome.

`IntPr.inst_step_tac` i is like `safe_step_tac`, but allows unknowns to be instantiated.

`IntPr.step_tac` i tries `safe_tac` or `inst_step_tac`, or applies an unsafe rule. This is the basic step of the intuitionistic proof procedure.

`IntPr.fast_tac` i applies `step_tac`, using depth-first search, to solve subgoal i.

```
excluded_middle    ~P | P

disjCI    (~Q ==> P) ==> P|Q
exCI      (ALL x. ~P(x) ==> P(a)) ==> EX 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
```

Figure 2.4: Derived rules for classical logic

`IntPr.best_tac` i applies `step_tac`, using best-first search (guided by the size of the proof state) to solve subgoal i.

Here are some of the theorems that `IntPr.fast_tac` proves automatically. The latter three date from *Principia Mathematica* (*11.53, *11.55, *11.61) [49].

```
~~P & ~~(P --> Q) --> ~~Q
(ALL x y. P(x) --> Q(y)) <-> ((EX x. P(x)) --> (ALL y. Q(y)))
(EX x y. P(x) & Q(x,y)) <-> (EX x. P(x) & (EX y. Q(x,y)))
(EX y. ALL x. P(x) --> Q(x,y)) --> (ALL x. P(x) --> (EX y. Q(x,y)))
```

2.4 Classical proof procedures

The classical theory, `FOL`, consists of intuitionistic logic plus the rule

$$\frac{\begin{matrix}[\neg P]\\ \vdots \\ P\end{matrix}}{P} \qquad (\textit{classical})$$

Natural deduction in classical logic is not really all that natural. `FOL` derives classical introduction rules for $\lor$ and $\exists$, as well as classical elimination rules for $\to$ and $\leftrightarrow$, and the swap rule (see Fig. 2.4).

The classical reasoner is installed. Tactics such as `Blast_tac` and `Best_tac` use the default claset (`!claset`), which works for most purposes. Named clasets include `prop_cs`, which includes the propositional rules, and `FOL_cs`, which also includes quantifier rules. See the file `FOL/cladata.ML` for lists of the classical rules, and the *Reference Manual* for more discussion of classical proof methods.

2.5 An intuitionistic example

Here is a session similar to one in *Logic and Computation* [32, pages 222–3]. Isabelle treats quantifiers differently from LCF-based theorem provers such as

HOL. The proof begins by entering the goal in intuitionistic logic, then applying the rule $(\rightarrow I)$.

```
goal IFOL.thy "(EX y. ALL x. Q(x,y)) -->  (ALL x. EX y. Q(x,y))";
  Level 0
  (EX y. ALL x. Q(x,y)) --> (ALL x. EX y. Q(x,y))
   1. (EX y. ALL x. Q(x,y)) --> (ALL x. EX y. Q(x,y))
by (resolve_tac [impI] 1);
  Level 1
  (EX y. ALL x. Q(x,y)) --> (ALL x. EX y. Q(x,y))
   1. EX y. ALL x. Q(x,y) ==> ALL x. EX y. Q(x,y)
```

In this example, we shall never have more than one subgoal. Applying $(\rightarrow I)$ replaces `-->` by `==>`, making $\exists y . \forall x . Q(x, y)$ an assumption. We have the choice of $(\exists E)$ and $(\forall I)$; let us try the latter.

```
by (resolve_tac [allI] 1);
  Level 2
  (EX y. ALL x. Q(x,y)) --> (ALL x. EX y. Q(x,y))
   1. !!x. EX y. ALL x. Q(x,y) ==> EX y. Q(x,y)
```

Applying $(\forall I)$ replaces the `ALL x` by `!!x`, changing the universal quantifier from object $(\forall)$ to meta $(\bigwedge)$. The bound variable is a **parameter** of the subgoal. We now must choose between $(\exists I)$ and $(\exists E)$. What happens if the wrong rule is chosen?

```
by (resolve_tac [exI] 1);
  Level 3
  (EX y. ALL x. Q(x,y)) --> (ALL x. EX y. Q(x,y))
   1. !!x. EX y. ALL x. Q(x,y) ==> Q(x,?y2(x))
```

The new subgoal 1 contains the function variable `?y2`. Instantiating `?y2` can replace `?y2(x)` by a term containing `x`, even though `x` is a bound variable. Now we analyse the assumption $\exists y . \forall x . Q(x, y)$ using elimination rules:

```
by (eresolve_tac [exE] 1);
  Level 4
  (EX y. ALL x. Q(x,y)) --> (ALL x. EX y. Q(x,y))
   1. !!x y. ALL x. Q(x,y) ==> Q(x,?y2(x))
```

Applying $(\exists E)$ has produced the parameter `y` and stripped the existential quantifier from the assumption. But the subgoal is unprovable: there is no way to unify `?y2(x)` with the bound variable `y`. Using `choplev` we can return to the critical point. This time we apply $(\exists E)$:

```
choplev 2;
  Level 2
  (EX y. ALL x. Q(x,y)) --> (ALL x. EX y. Q(x,y))
   1. !!x. EX y. ALL x. Q(x,y) ==> EX y. Q(x,y)
```

```
by (eresolve_tac [exE] 1);
  Level 3
  (EX y. ALL x. Q(x,y)) --> (ALL x. EX y. Q(x,y))
   1. !!x y. ALL x. Q(x,y) ==> EX y. Q(x,y)
```

We now have two parameters and no scheme variables. Applying $(\exists I)$ and $(\forall E)$ produces two scheme variables, which are applied to those parameters. Parameters should be produced early, as this example demonstrates.

```
by (resolve_tac [exI] 1);
  Level 4
  (EX y. ALL x. Q(x,y)) --> (ALL x. EX y. Q(x,y))
   1. !!x y. ALL x. Q(x,y) ==> Q(x,?y3(x,y))
by (eresolve_tac [allE] 1);
  Level 5
  (EX y. ALL x. Q(x,y)) --> (ALL x. EX y. Q(x,y))
   1. !!x y. Q(?x4(x,y),y) ==> Q(x,?y3(x,y))
```

The subgoal has variables `?y3` and `?x4` applied to both parameters. The obvious projection functions unify `?x4(x,y)` with `x` and `?y3(x,y)` with `y`.

```
by (assume_tac 1);
  Level 6
  (EX y. ALL x. Q(x,y)) --> (ALL x. EX y. Q(x,y))
  No subgoals!
```

The theorem was proved in six tactic steps, not counting the abandoned ones. But proof checking is tedious; `IntPr.fast_tac` proves the theorem in one step.

```
goal IFOL.thy "(EX y. ALL x. Q(x,y)) -->  (ALL x. EX y. Q(x,y))";
  Level 0
  (EX y. ALL x. Q(x,y)) --> (ALL x. EX y. Q(x,y))
   1. (EX y. ALL x. Q(x,y)) --> (ALL x. EX y. Q(x,y))
by (IntPr.fast_tac 1);
  Level 1
  (EX y. ALL x. Q(x,y)) --> (ALL x. EX y. Q(x,y))
  No subgoals!
```

2.6 An example of intuitionistic negation

The following example demonstrates the specialized forms of implication elimination. Even propositional formulae can be difficult to prove from the basic rules; the specialized rules help considerably.

Propositional examples are easy to invent. As Dummett notes [11, page 28], $\neg P$ is classically provable if and only if it is intuitionistically provable; therefore,

P is classically provable if and only if $\neg\neg P$ is intuitionistically provable.[1] Proving $\neg\neg P$ intuitionistically is much harder than proving P classically.

Our example is the double negation of the classical tautology $(P \rightarrow Q) \vee (Q \rightarrow P)$. When stating the goal, we command Isabelle to expand negations to implications using the definition $\neg P \equiv P \rightarrow \bot$. This allows use of the special implication rules.

```
goalw IFOL.thy [not_def] "~ ~ ((P-->Q) | (Q-->P))";
  Level 0
  ~ ~ ((P --> Q) | (Q --> P))
   1. ((P --> Q) | (Q --> P) --> False) --> False
```

The first step is trivial.

```
by (resolve_tac [impI] 1);
  Level 1
  ~ ~ ((P --> Q) | (Q --> P))
   1. (P --> Q) | (Q --> P) --> False ==> False
```

By $(\rightarrow E)$ it would suffice to prove $(P \rightarrow Q) \vee (Q \rightarrow P)$, but that formula is not a theorem of intuitionistic logic. Instead we apply the specialized implication rule `disj_impE`. It splits the assumption into two assumptions, one for each disjunct.

```
by (eresolve_tac [disj_impE] 1);
  Level 2
  ~ ~ ((P --> Q) | (Q --> P))
   1. [| (P --> Q) --> False; (Q --> P) --> False |] ==> False
```

We cannot hope to prove $P \rightarrow Q$ or $Q \rightarrow P$ separately, but their negations are inconsistent. Applying `imp_impE` breaks down the assumption $\neg(P \rightarrow Q)$, asking to show Q while providing new assumptions P and $\neg Q$.

```
by (eresolve_tac [imp_impE] 1);
  Level 3
  ~ ~ ((P --> Q) | (Q --> P))
   1. [| (Q --> P) --> False; P; Q --> False |] ==> Q
   2. [| (Q --> P) --> False; False |] ==> False
```

Subgoal 2 holds trivially; let us ignore it and continue working on subgoal 1. Thanks to the assumption P, we could prove $Q \rightarrow P$; applying `imp_impE` is simpler.

```
by (eresolve_tac [imp_impE] 1);
  Level 4
  ~ ~ ((P --> Q) | (Q --> P))
   1. [| P; Q --> False; Q; P --> False |] ==> P
   2. [| P; Q --> False; False |] ==> Q
   3. [| (Q --> P) --> False; False |] ==> False
```

[1]Of course this holds only for propositional logic, not if P is allowed to contain quantifiers.

The three subgoals are all trivial.

```
by (REPEAT (eresolve_tac [FalseE] 2));
  Level 5
  ~ ~ ((P --> Q) | (Q --> P))
   1. [| P; Q --> False; Q; P --> False |] ==> P
by (assume_tac 1);
  Level 6
  ~ ~ ((P --> Q) | (Q --> P))
  No subgoals!
```

This proof is also trivial for `IntPr.fast_tac`.

2.7 A classical example

To illustrate classical logic, we shall prove the theorem $\exists y\,.\,\forall x\,.\,P(y) \rightarrow P(x)$. Informally, the theorem can be proved as follows. Choose y such that $\neg P(y)$, if such exists; otherwise $\forall x\,.\,P(x)$ is true. Either way the theorem holds.

The formal proof does not conform in any obvious way to the sketch given above. The key inference is the first one, `exCI`; this classical version of $(\exists I)$ allows multiple instantiation of the quantifier.

```
goal FOL.thy "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 [exCI] 1);
  Level 1
  EX y. ALL x. P(y) --> P(x)
   1. ALL y. ~ (ALL x. P(y) --> P(x)) ==> ALL x. P(?a) --> P(x)
```

We can either exhibit a term `?a` to satisfy the conclusion of subgoal 1, or produce a contradiction from the assumption. The next steps are routine.

```
by (resolve_tac [allI] 1);
  Level 2
  EX y. ALL x. P(y) --> P(x)
   1. !!x. ALL y. ~ (ALL x. P(y) --> P(x)) ==> P(?a) --> P(x)
by (resolve_tac [impI] 1);
  Level 3
  EX y. ALL x. P(y) --> P(x)
   1. !!x. [| ALL y. ~ (ALL x. P(y) --> P(x)); P(?a) |] ==> P(x)
```

By the duality between ∃ and ∀, applying (∀E) in effect applies (∃I) again.

```
by (eresolve_tac [allE] 1);
  Level 4
  EX y. ALL x. P(y) --> P(x)
   1. !!x. [| P(?a); ~ (ALL xa. P(?y3(x)) --> P(xa)) |] ==> P(x)
```

In classical logic, a negated assumption is equivalent to a conclusion. To get this effect, we create a swapped version of (∀I) and apply it using `eresolve_tac`; we could equivalently have applied (∀I) using `swap_res_tac`.

```
allI RSN (2,swap);
  val it = "[| ~ (ALL x. ?P1(x)); !!x. ~ ?Q ==> ?P1(x) |] ==> ?Q" : thm
by (eresolve_tac [it] 1);
  Level 5
  EX y. ALL x. P(y) --> P(x)
   1. !!x xa. [| P(?a); ~ P(x) |] ==> P(?y3(x)) --> P(xa)
```

The previous conclusion, `P(x)`, has become a negated assumption.

```
by (resolve_tac [impI] 1);
  Level 6
  EX y. ALL x. P(y) --> P(x)
   1. !!x xa. [| P(?a); ~ P(x); P(?y3(x)) |] ==> P(xa)
```

The subgoal has three assumptions. We produce a contradiction between the assumptions `~P(x)` and `P(?y3(x))`. The proof never instantiates the unknown `?a`.

```
by (eresolve_tac [notE] 1);
  Level 7
  EX y. ALL x. P(y) --> P(x)
   1. !!x xa. [| P(?a); P(?y3(x)) |] ==> P(x)

by (assume_tac 1);
  Level 8
  EX y. ALL x. P(y) --> P(x)
  No subgoals!
```

The civilised way to prove this theorem is through `deepen_tac`, which automatically uses the classical version of (∃I):

```
goal FOL.thy "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 (Deepen_tac 0 1);
  Depth = 0
  Depth = 2
  Level 1
  EX y. ALL x. P(y) --> P(x)
  No subgoals!
```

If this theorem seems counterintuitive, then perhaps you are an intuitionist. In constructive logic, proving $\exists y\,.\,\forall x\,.\,P(y) \rightarrow P(x)$ requires exhibiting a particular term t such that $\forall x.P(t) \rightarrow P(x)$, which we cannot do without further knowledge about P.

2.8 Derived rules and the classical tactics

Classical first-order logic can be extended with the propositional connective $\mathit{if}(P, Q, R)$, where

$$\mathit{if}(P, Q, R) \equiv P \land Q \lor \neg P \land R. \qquad (\mathit{if})$$

Theorems about *if* can be proved by treating this as an abbreviation, replacing $\mathit{if}(P, Q, R)$ by $P \land Q \lor \neg P \land R$ in subgoals. But this duplicates P, causing an exponential blowup and an unreadable formula. Introducing further abbreviations makes the problem worse.

Natural deduction demands rules that introduce and eliminate $\mathit{if}(P, Q, R)$ directly, without reference to its definition. The simple identity

$$\mathit{if}(P, Q, R) \;\leftrightarrow\; (P \rightarrow Q) \land (\neg P \rightarrow R)$$

suggests that the *if*-introduction rule should be

$$\frac{\begin{matrix}[P] \\ \vdots \\ Q\end{matrix} \qquad \begin{matrix}[\neg P] \\ \vdots \\ R\end{matrix}}{\mathit{if}(P, Q, R)}\;(\mathit{if}\ I)$$

The *if*-elimination rule reflects the definition of $\mathit{if}(P, Q, R)$ and the elimination rules for $\lor$ and $\land$.

$$\frac{\mathit{if}(P, Q, R) \qquad \begin{matrix}[P, Q] \\ \vdots \\ S\end{matrix} \qquad \begin{matrix}[\neg P, R] \\ \vdots \\ S\end{matrix}}{S}\;(\mathit{if}\ E)$$

Having made these plans, we get down to work with Isabelle. The theory of classical logic, `FOL`, is extended with the constant $\mathit{if} :: [o, o, o] \Rightarrow o$. The axiom `if_def` asserts the equation (if).

```
If = FOL +
consts  if     :: [o,o,o]=>o
rules   if_def "if(P,Q,R) == P&Q | ~P&R"
end
```

The derivations of the introduction and elimination rules demonstrate the methods for rewriting with definitions. Classical reasoning is required, so we use `blast_tac`.

2.8.1 Deriving the introduction rule

The introduction rule, given the premises $P \Longrightarrow Q$ and $\neg P \Longrightarrow R$, concludes $\mathit{if}(P, Q, R)$. We propose the conclusion as the main goal using `goalw`, which uses `if_def` to rewrite occurrences of *if* in the subgoal.

```
val prems = goalw If.thy [if_def]
    "[| P ==> Q; ~ P ==> R |] ==> if(P,Q,R)";
  Level 0
  if(P,Q,R)
   1. P & Q | ~ P & R
```

The premises (bound to the ML variable `prems`) are passed as introduction rules to `blast_tac`. Remember that `!claset` refers to the default classical set.

```
by (blast_tac (!claset addIs prems) 1);
  Level 1
  if(P,Q,R)
  No subgoals!
qed "ifI";
```

2.8.2 Deriving the elimination rule

The elimination rule has three premises, two of which are themselves rules. The conclusion is simply S.

```
val major::prems = goalw If.thy [if_def]
   "[| if(P,Q,R);  [| P; Q |] ==> S; [| ~ P; R |] ==> S |] ==> S";
  Level 0
  S
   1. S
```

The major premise contains an occurrence of *if*, but the version returned by `goalw` (and bound to the ML variable `major`) has the definition expanded. Now `cut_facts_tac` inserts `major` as an assumption in the subgoal, so that `blast_tac` can break it down.

```
by (cut_facts_tac [major] 1);
  Level 1
  S
   1. P & Q | ~ P & R ==> S
by (blast_tac (!claset addIs prems) 1);
  Level 2
  S
  No subgoals!
qed "ifE";
```

As you may recall from *Introduction to Isabelle*, there are other ways of treating definitions when deriving a rule. We can start the proof using `goal`, which does not expand definitions, instead of `goalw`. We can use `rew_tac` to expand definitions in the subgoals — perhaps after calling `cut_facts_tac` to insert the rule's premises. We can use `rewrite_rule`, which is a meta-inference rule, to expand definitions in the premises directly.

2.8.3 Using the derived rules

The rules just derived have been saved with the ML names `ifI` and `ifE`. They permit natural proofs of theorems such as the following:

$$\begin{aligned} \mathit{if}(P, \mathit{if}(Q, A, B), \mathit{if}(Q, C, D)) &\leftrightarrow \mathit{if}(Q, \mathit{if}(P, A, C), \mathit{if}(P, B, D)) \\ \mathit{if}(\mathit{if}(P, Q, R), A, B) &\leftrightarrow \mathit{if}(P, \mathit{if}(Q, A, B), \mathit{if}(R, A, B)) \end{aligned}$$

Proofs also require the classical reasoning rules and the $\leftrightarrow$ introduction rule (called `iffI`: do not confuse with `ifI`).

To display the *if*-rules in action, let us analyse a proof step by step.

```
goal If.thy
    "if(P, if(Q,A,B), if(Q,C,D)) <-> if(Q, if(P,A,C), if(P,B,D))";
  Level 0
  if(P,if(Q,A,B),if(Q,C,D)) <-> if(Q,if(P,A,C),if(P,B,D))
   1. if(P,if(Q,A,B),if(Q,C,D)) <-> if(Q,if(P,A,C),if(P,B,D))
by (resolve_tac [iffI] 1);
  Level 1
  if(P,if(Q,A,B),if(Q,C,D)) <-> if(Q,if(P,A,C),if(P,B,D))
   1. if(P,if(Q,A,B),if(Q,C,D)) ==> if(Q,if(P,A,C),if(P,B,D))
   2. if(Q,if(P,A,C),if(P,B,D)) ==> if(P,if(Q,A,B),if(Q,C,D))
```

The *if*-elimination rule can be applied twice in succession.

```
by (eresolve_tac [ifE] 1);
  Level 2
  if(P,if(Q,A,B),if(Q,C,D)) <-> if(Q,if(P,A,C),if(P,B,D))
   1. [| P; if(Q,A,B) |] ==> if(Q,if(P,A,C),if(P,B,D))
   2. [| ~ P; if(Q,C,D) |] ==> if(Q,if(P,A,C),if(P,B,D))
   3. if(Q,if(P,A,C),if(P,B,D)) ==> if(P,if(Q,A,B),if(Q,C,D))
by (eresolve_tac [ifE] 1);
  Level 3
  if(P,if(Q,A,B),if(Q,C,D)) <-> if(Q,if(P,A,C),if(P,B,D))
   1. [| P; Q; A |] ==> if(Q,if(P,A,C),if(P,B,D))
   2. [| P; ~ Q; B |] ==> if(Q,if(P,A,C),if(P,B,D))
   3. [| ~ P; if(Q,C,D) |] ==> if(Q,if(P,A,C),if(P,B,D))
   4. if(Q,if(P,A,C),if(P,B,D)) ==> if(P,if(Q,A,B),if(Q,C,D))
```

In the first two subgoals, all assumptions have been reduced to atoms. Now *if*-introduction can be applied. Observe how the *if*-rules break down occurrences of *if* when they become the outermost connective.

```
by (resolve_tac [ifI] 1);
  Level 4
  if(P,if(Q,A,B),if(Q,C,D)) <-> if(Q,if(P,A,C),if(P,B,D))
   1. [| P; Q; A; Q |] ==> if(P,A,C)
   2. [| P; Q; A; ~ Q |] ==> if(P,B,D)
   3. [| P; ~ Q; B |] ==> if(Q,if(P,A,C),if(P,B,D))
   4. [| ~ P; if(Q,C,D) |] ==> if(Q,if(P,A,C),if(P,B,D))
   5. if(Q,if(P,A,C),if(P,B,D)) ==> if(P,if(Q,A,B),if(Q,C,D))
by (resolve_tac [ifI] 1);
  Level 5
  if(P,if(Q,A,B),if(Q,C,D)) <-> if(Q,if(P,A,C),if(P,B,D))
   1. [| P; Q; A; Q; P |] ==> A
   2. [| P; Q; A; Q; ~ P |] ==> C
   3. [| P; Q; A; ~ Q |] ==> if(P,B,D)
   4. [| P; ~ Q; B |] ==> if(Q,if(P,A,C),if(P,B,D))
   5. [| ~ P; if(Q,C,D) |] ==> if(Q,if(P,A,C),if(P,B,D))
   6. if(Q,if(P,A,C),if(P,B,D)) ==> if(P,if(Q,A,B),if(Q,C,D))
```

Where do we stand? The first subgoal holds by assumption; the second and third, by contradiction. This is getting tedious. We could use the classical reasoner, but first let us extend the default claset with the derived rules for *if*.

```
AddSIs [ifI];
AddSEs [ifE];
```

Now we can revert to the initial proof state and let `blast_tac` solve it.

```
choplev 0;
  Level 0
  if(P,if(Q,A,B),if(Q,C,D)) <-> if(Q,if(P,A,C),if(P,B,D))
   1. if(P,if(Q,A,B),if(Q,C,D)) <-> if(Q,if(P,A,C),if(P,B,D))
by (Blast_tac 1);
  Level 1
  if(P,if(Q,A,B),if(Q,C,D)) <-> if(Q,if(P,A,C),if(P,B,D))
  No subgoals!
```

This tactic also solves the other example.

```
goal If.thy "if(if(P,Q,R), A, B) <-> if(P, if(Q,A,B), if(R,A,B))";
  Level 0
  if(if(P,Q,R),A,B) <-> if(P,if(Q,A,B),if(R,A,B))
   1. if(if(P,Q,R),A,B) <-> if(P,if(Q,A,B),if(R,A,B))
by (Blast_tac 1);
  Level 1
  if(if(P,Q,R),A,B) <-> if(P,if(Q,A,B),if(R,A,B))
  No subgoals!
```

2.8.4 Derived rules versus definitions

Dispensing with the derived rules, we can treat *if* as an abbreviation, and let `blast_tac` prove the expanded formula. Let us redo the previous proof:

```
choplev 0;
  Level 0
  if(if(P,Q,R),A,B) <-> if(P,if(Q,A,B),if(R,A,B))
   1. if(if(P,Q,R),A,B) <-> if(P,if(Q,A,B),if(R,A,B))
```

This time, simply unfold using the definition of *if*:

```
by (rewtac if_def);
  Level 1
  if(if(P,Q,R),A,B) <-> if(P,if(Q,A,B),if(R,A,B))
   1. (P & Q | ~ P & R) & A | ~ (P & Q | ~ P & R) & B <->
      P & (Q & A | ~ Q & B) | ~ P & (R & A | ~ R & B)
```

We are left with a subgoal in pure first-order logic, which is why the classical reasoner can prove it given `FOL_cs` alone. (We could, of course, have used `Blast_tac`.)

```
by (blast_tac FOL_cs 1);
  Level 2
  if(if(P,Q,R),A,B) <-> if(P,if(Q,A,B),if(R,A,B))
  No subgoals!
```

Expanding definitions reduces the extended logic to the base logic. This approach has its merits — especially if the prover for the base logic is good — but can be slow. In these examples, proofs using the default claset (which includes the derived rules) run about six times faster than proofs using `FOL_cs`.

Expanding definitions also complicates error diagnosis. Suppose we are having difficulties in proving some goal. If by expanding definitions we have made it unreadable, then we have little hope of diagnosing the problem.

Attempts at program verification often yield invalid assertions. Let us try to prove one:

```
goal If.thy "if(if(P,Q,R), A, B) <-> if(P, if(Q,A,B), if(R,B,A))";
  Level 0
  if(if(P,Q,R),A,B) <-> if(P,if(Q,A,B),if(R,B,A))
   1. if(if(P,Q,R),A,B) <-> if(P,if(Q,A,B),if(R,B,A))
by (Blast_tac 1);
  by: tactic failed
```

This failure message is uninformative, but we can get a closer look at the situation by applying `Step_tac`.

```
by (REPEAT (Step_tac 1));
  Level 1
  if(if(P,Q,R),A,B) <-> if(P,if(Q,A,B),if(R,B,A))
   1. [| A; ~ P; R; ~ P; R |] ==> B
   2. [| B; ~ P; ~ R; ~ P; ~ R |] ==> A
   3. [| ~ P; R; B; ~ P; R |] ==> A
   4. [| ~ P; ~ R; A; ~ B; ~ P |] ==> R
```

Subgoal 1 is unprovable and yields a countermodel: P and B are false while R and A are true. This truth assignment reduces the main goal to *true* $\leftrightarrow$ *false*, which is of course invalid.

We can repeat this analysis by expanding definitions, using just the rules of `FOL`:

```
choplev 0;
  Level 0
  if(if(P,Q,R),A,B) <-> if(P,if(Q,A,B),if(R,B,A))
   1. if(if(P,Q,R),A,B) <-> if(P,if(Q,A,B),if(R,B,A))
by (rewtac if_def);
  Level 1
  if(if(P,Q,R),A,B) <-> if(P,if(Q,A,B),if(R,B,A))
   1. (P & Q | ~ P & R) & A | ~ (P & Q | ~ P & R) & B <->
      P & (Q & A | ~ Q & B) | ~ P & (R & B | ~ R & A)
by (blast_tac FOL_cs 1);
  by: tactic failed
```

Again we apply `step_tac`:

```
by (REPEAT (step_tac FOL_cs 1));
  Level 2
  if(if(P,Q,R),A,B) <-> if(P,if(Q,A,B),if(R,B,A))
   1. [| A; ~ P; R; ~ P; R; ~ False |] ==> B
   2. [| A; ~ P; R; R; ~ False; ~ B; ~ B |] ==> Q
   3. [| B; ~ P; ~ R; ~ P; ~ A |] ==> R
   4. [| B; ~ P; ~ R; ~ Q; ~ A |] ==> R
   5. [| B; ~ R; ~ P; ~ A; ~ R; Q; ~ False |] ==> A
   6. [| ~ P; R; B; ~ P; R; ~ False |] ==> A
   7. [| ~ P; ~ R; A; ~ B; ~ R |] ==> P
   8. [| ~ P; ~ R; A; ~ B; ~ R |] ==> Q
```

Subgoal 1 yields the same countermodel as before. But each proof step has taken six times as long, and the final result contains twice as many subgoals.

Expanding definitions causes a great increase in complexity. This is why the classical prover has been designed to accept derived rules.

Chapter 3

Zermelo-Fraenkel Set Theory

The theory `ZF` implements Zermelo-Fraenkel set theory [17, 45] as an extension of `FOL`, classical first-order logic. The theory includes a collection of derived natural deduction rules, for use with Isabelle's classical reasoner. Much of it is based on the work of Noël [28].

A tremendous amount of set theory has been formally developed, including the basic properties of relations, functions, ordinals and cardinals. Significant results have been proved, such as the Schröder-Bernstein Theorem, the Wellordering Theorem and a version of Ramsey's Theorem. General methods have been developed for solving recursion equations over monotonic functors; these have been applied to yield constructions of lists, trees, infinite lists, etc. The Recursion Theorem has been proved, admitting recursive definitions of functions over well-founded relations. Thus, we may even regard set theory as a computational logic, loosely inspired by Martin-Löf's Type Theory.

Because `ZF` is an extension of `FOL`, it provides the same packages, namely `hyp_subst_tac`, the simplifier, and the classical reasoner. The default simpset and claset are usually satisfactory. Named simpsets include `ZF_ss` (basic set theory rules) and `rank_ss` (for proving termination of well-founded recursion). Named clasets include `ZF_cs` (basic set theory) and `le_cs` (useful for reasoning about the relations $<$ and $\leq$).

`ZF` has a flexible package for handling inductive definitions, such as inference systems, and datatype definitions, such as lists and trees. Moreover it handles coinductive definitions, such as bisimulation relations, and codatatype definitions, such as streams. There is a paper [34] describing the package, but its examples use an obsolete declaration syntax. Please consult the version of the paper distributed with Isabelle.

Recent reports [33, 35] describe `ZF` less formally than this chapter. Isabelle employs a novel treatment of non-well-founded data structures within the standard ZF axioms including the Axiom of Foundation [40].

3.1 Which version of axiomatic set theory?

The two main axiom systems for set theory are Bernays-Gödel (BG) and Zermelo-Fraenkel (ZF). Resolution theorem provers can use BG because it is finite [4, 43]. ZF does not have a finite axiom system because of its Axiom Scheme of Replacement. This makes it awkward to use with many theorem provers, since instances of the axiom scheme have to be invoked explicitly. Since Isabelle has no difficulty with axiom schemes, we may adopt either axiom system.

These two theories differ in their treatment of **classes**, which are collections that are 'too big' to be sets. The class of all sets, V, cannot be a set without admitting Russell's Paradox. In BG, both classes and sets are individuals; $x \in V$ expresses that x is a set. In ZF, all variables denote sets; classes are identified with unary predicates. The two systems define essentially the same sets and classes, with similar properties. In particular, a class cannot belong to another class (let alone a set).

Modern set theorists tend to prefer ZF because they are mainly concerned with sets, rather than classes. BG requires tiresome proofs that various collections are sets; for instance, showing $x \in \{x\}$ requires showing that x is a set.

3.2 The syntax of set theory

The language of set theory, as studied by logicians, has no constants. The traditional axioms merely assert the existence of empty sets, unions, powersets, etc.; this would be intolerable for practical reasoning. The Isabelle theory declares constants for primitive sets. It also extends `FOL` with additional syntax for finite sets, ordered pairs, comprehension, general union/intersection, general sums/products, and bounded quantifiers. In most other respects, Isabelle implements precisely Zermelo-Fraenkel set theory.

Figure 3.1 lists the constants and infixes of `ZF`, while Figure 3.2 presents the syntax translations. Finally, Figure 3.3 presents the full grammar for set theory, including the constructs of `FOL`.

Local abbreviations can be introduced by a `let` construct whose syntax appears in Fig. 3.3. Internally it is translated into the constant `Let`. It can be expanded by rewriting with its definition, `Let_def`.

Apart from `let`, set theory does not use polymorphism. All terms in `ZF` have type i, which is the type of individuals and has class `term`. The type of first-order formulae, remember, is o.

Infix operators include binary union and intersection ($A \cup B$ and $A \cap B$), set difference ($A - B$), and the subset and membership relations. Note that a`~:`b is translated to $\neg(a \in b)$. The union and intersection operators ($\bigcup A$ and $\bigcap A$) form the union or intersection of a set of sets; $\bigcup A$ means the same as $\bigcup_{x \in A} x$. Of these operators, only $\bigcup A$ is primitive.

The constant `Upair` constructs unordered pairs; thus `Upair(`A`,`B`)` denotes the set $\{A, B\}$ and `Upair(`A`,`A`)` denotes the singleton $\{A\}$. General union is used to define binary union. The Isabelle version goes on to define the constant

name	*meta-type*	*description*
`Let`	$[\alpha, \alpha \Rightarrow \beta] \Rightarrow \beta$	let binder
`0`	i	empty set
`cons`	$[i, i] \Rightarrow i$	finite set constructor
`Upair`	$[i, i] \Rightarrow i$	unordered pairing
`Pair`	$[i, i] \Rightarrow i$	ordered pairing
`Inf`	i	infinite set
`Pow`	$i \Rightarrow i$	powerset
`Union Inter`	$i \Rightarrow i$	set union/intersection
`split`	$[[i, i] \Rightarrow i, i] \Rightarrow i$	generalized projection
`fst snd`	$i \Rightarrow i$	projections
`converse`	$i \Rightarrow i$	converse of a relation
`succ`	$i \Rightarrow i$	successor
`Collect`	$[i, i \Rightarrow o] \Rightarrow i$	separation
`Replace`	$[i, [i, i] \Rightarrow o] \Rightarrow i$	replacement
`PrimReplace`	$[i, [i, i] \Rightarrow o] \Rightarrow i$	primitive replacement
`RepFun`	$[i, i \Rightarrow i] \Rightarrow i$	functional replacement
`Pi Sigma`	$[i, i \Rightarrow i] \Rightarrow i$	general product/sum
`domain`	$i \Rightarrow i$	domain of a relation
`range`	$i \Rightarrow i$	range of a relation
`field`	$i \Rightarrow i$	field of a relation
`Lambda`	$[i, i \Rightarrow i] \Rightarrow i$	λ-abstraction
`restrict`	$[i, i] \Rightarrow i$	restriction of a function
`The`	$[i \Rightarrow o] \Rightarrow i$	definite description
`if`	$[o, i, i] \Rightarrow i$	conditional
`Ball Bex`	$[i, i \Rightarrow o] \Rightarrow o$	bounded quantifiers

CONSTANTS

symbol	*meta-type*	*priority*	*description*
``` `` ```	$[i, i] \Rightarrow i$	Left 90	image
``` -`` ```	$[i, i] \Rightarrow i$	Left 90	inverse image
`` ` ``	$[i, i] \Rightarrow i$	Left 90	application
`Int`	$[i, i] \Rightarrow i$	Left 70	intersection ($\cap$)
`Un`	$[i, i] \Rightarrow i$	Left 65	union ($\cup$)
`-`	$[i, i] \Rightarrow i$	Left 65	set difference ($-$)
`:`	$[i, i] \Rightarrow o$	Left 50	membership ($\in$)
`<=`	$[i, i] \Rightarrow o$	Left 50	subset ($\subseteq$)

INFIXES

Figure 3.1: Constants of `ZF`

external	*internal*	*description*
`a ~: b`	`~(a : b)`	negated membership
`{`a_1`, ..., `a_n`}`	`cons(`a_1`,...,cons(`a_n`,0))`	finite set
`<`a_1`, ..., `a_{n-1}`, `a_n`>`	`Pair(`a_1`,...,Pair(`a_{n-1}`,`a_n`)...)`	ordered n-tuple
`{`x:A . $P[x]$`}`	`Collect(`A`,`$\lambda x . P[x]$`)`	separation
`{`y . x:A, $Q[x,y]$`}`	`Replace(`A`,`$\lambda x\, y . Q[x,y]$`)`	replacement
`{`$b[x]$. x:A`}`	`RepFun(`A`,`$\lambda x . b[x]$`)`	functional replacement
`INT` x:A . $B[x]$	`Inter({`$B[x]$. x:A`})`	general intersection
`UN` x:A . $B[x]$	`Union({`$B[x]$. x:A`})`	general union
`PROD` x:A . $B[x]$	`Pi(`A`,`$\lambda x . B[x]$`)`	general product
`SUM` x:A . $B[x]$	`Sigma(`A`,`$\lambda x . B[x]$`)`	general sum
A `->` B	`Pi(`A`,`$\lambda x . B$`)`	function space
A `*` B	`Sigma(`A`,`$\lambda x . B$`)`	binary product
`THE` x . $P[x]$	`The(`$\lambda x . P[x]$`)`	definite description
`lam` x:A . $b[x]$	`Lambda(`A`,`$\lambda x . b[x]$`)`	λ-abstraction
`ALL` x:A . $P[x]$	`Ball(`A`,`$\lambda x . P[x]$`)`	bounded $\forall$
`EX` x:A . $P[x]$	`Bex(`A`,`$\lambda x . P[x]$`)`	bounded $\exists$

Figure 3.2: Translations for `ZF`

`cons`:

$$\begin{aligned} A \cup B &\equiv \bigcup(\texttt{Upair}(A, B)) \\ \texttt{cons}(a, B) &\equiv \texttt{Upair}(a, a) \cup B \end{aligned}$$

The $\{a_1, \ldots\}$ notation abbreviates finite sets constructed in the obvious manner using `cons` and $\emptyset$ (the empty set):

$$\{a, b, c\} \equiv \texttt{cons}(a, \texttt{cons}(b, \texttt{cons}(c, \emptyset)))$$

The constant `Pair` constructs ordered pairs, as in `Pair(`a`,`b`)`. Ordered pairs may also be written within angle brackets, as `<`a`,`b`>`. The n-tuple `<`a_1`,...,`a_{n-1}`,`a_n`>` abbreviates the nest of pairs

Pair(a_1,...,Pair(a_{n-1},a_n)...).

In `ZF`, a function is a set of pairs. A `ZF` function f is simply an individual as far as Isabelle is concerned: its Isabelle type is i, not say $i \Rightarrow i$. The infix operator ` denotes the application of a function set to its argument; we must write $f`x$, not $f(x)$. The syntax for image is $f``A$ and that for inverse image is $f-``A$.

3.3 Binding operators

The constant `Collect` constructs sets by the principle of **separation**. The syntax for separation is `{`x:A.$P[x]$`}`, where $P[x]$ is a formula that may contain free

```
   term  =  expression of type i
         |  let id = term; ... ; id = term in term
         |  { term (,term)* }
         |  < term (,term)* >
         |  { id:term . formula }
         |  { id . id:term, formula }
         |  { term . id:term }
         |  term `` term
         |  term -`` term
         |  term ` term
         |  term * term
         |  term Int term
         |  term Un term
         |  term - term
         |  term -> term
         |  THE  id . formula
         |  lam  id:term . term
         |  INT  id:term . term
         |  UN   id:term . term
         |  PROD id:term . term
         |  SUM  id:term . term

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

Figure 3.3: Full grammar for ZF

occurrences of x. It abbreviates the set `Collect(`A`,`$\lambda x\,.\,P[x]$`)`, which consists of all $x \in A$ that satisfy $P[x]$. Note that `Collect` is an unfortunate choice of name: some set theories adopt a set-formation principle, related to replacement, called collection.

The constant `Replace` constructs sets by the principle of **replacement**. The syntax `{`$y\,.\,x\,{:}\,A$`,`$Q[x,y]$`}` denotes the set `Replace(`A`,`$\lambda x\, y\,.\,Q[x,y]$`)`, which consists of all y such that there exists $x \in A$ satisfying $Q[x,y]$. The Replacement Axiom has the condition that Q must be single-valued over A: for all $x \in A$ there exists at most one y satisfying $Q[x,y]$. A single-valued binary predicate is also called a **class function**.

The constant `RepFun` expresses a special case of replacement, where $Q[x,y]$ has the form $y = b[x]$. Such a Q is trivially single-valued, since it is just the graph of the meta-level function $\lambda x\,.\,b[x]$. The resulting set consists of all $b[x]$ for $x \in A$. This is analogous to the ML functional `map`, since it applies a function to every element of a set. The syntax is `{`$b[x]\,.\,x\,{:}\,A$`}`, which expands to `RepFun(`A`,`$\lambda x\,.\,b[x]$`)`.

General unions and intersections of indexed families of sets, namely $\bigcup_{x\in A} B[x]$ and $\bigcap_{x\in A} B[x]$, are written `UN` $x{:}A.B[x]$ and `INT` $x{:}A.B[x]$. Their meaning is expressed using `RepFun` as

$$\bigcup(\{B[x]\,.\,x\in A\}) \qquad \text{and} \qquad \bigcap(\{B[x]\,.\,x\in A\}).$$

General sums $\sum_{x\in A} B[x]$ and products $\prod_{x\in A} B[x]$ can be constructed in set theory, where $B[x]$ is a family of sets over A. They have as special cases $A \times B$ and $A \to B$, where B is simply a set. This is similar to the situation in Constructive Type Theory (set theory has 'dependent sets') and calls for similar syntactic conventions. The constants `Sigma` and `Pi` construct general sums and products. Instead of `Sigma(`A`,`B`)` and `Pi(`A`,`B`)` we may write `SUM` $x{:}A.B[x]$ and `PROD` $x{:}A.B[x]$. 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.

As mentioned above, whenever the axioms assert the existence and uniqueness of a set, Isabelle's set theory declares a constant for that set. These constants can express the **definite description** operator $\iota x\,.\,P[x]$, which stands for the unique a satisfying $P[a]$, if such exists. Since all terms in `ZF` denote something, a description is always meaningful, but we do not know its value unless $P[x]$ defines it uniquely. Using the constant `The`, we may write descriptions as `The(`$\lambda x\,.\,P[x]$`)` or use the syntax `THE` $x\,.\,P[x]$.

Function sets may be written in λ-notation; $\lambda x \in A\,.\,b[x]$ stands for the set of all pairs $\langle x, b[x]\rangle$ for $x \in A$. In order for this to be a set, the function's

[1]Unlike normal infix operators, `*` and `->` merely define abbreviations; there are no constants `op *` and `op ->`.

domain A must be given. Using the constant `Lambda`, we may express function sets as `Lambda(`A`,`$\lambda x\,.\,b[x]$`)` or use the syntax `lam` $x\!:\!A\,.\,b[x]$.

Isabelle's set theory defines two **bounded quantifiers**:

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

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` $x\!:\!A\,.\,P[x]$.

3.4 The Zermelo-Fraenkel axioms

The axioms appear in Fig. 3.4. They resemble those presented by Suppes [45]. Most of the theory consists of definitions. In particular, bounded quantifiers and the subset relation appear in other axioms. Object-level quantifiers and implications have been replaced by meta-level ones wherever possible, to simplify use of the axioms. See the file `ZF/ZF.thy` for details.

The traditional replacement axiom asserts

$$y\in \texttt{PrimReplace}(A,P)\leftrightarrow(\exists x\in A\,.\,P(x,y))$$

subject to the condition that $P(x,y)$ is single-valued for all $x\in A$. The Isabelle theory defines `Replace` to apply `PrimReplace` to the single-valued part of P, namely

$$(\exists! z\,.\,P(x,z))\land P(x,y).$$

Thus $y\in$ `Replace`(A,P) if and only if there is some x such that $P(x,-)$ holds uniquely for y. Because the equivalence is unconditional, `Replace` is much easier to use than `PrimReplace`; it defines the same set, if $P(x,y)$ is single-valued. The nice syntax for replacement expands to `Replace`.

Other consequences of replacement include functional replacement (`RepFun`) and definite descriptions (`The`). Axioms for separation (`Collect`) and unordered pairs (`Upair`) are traditionally assumed, but they actually follow from replacement [45, pages 237–8].

The definitions of general intersection, etc., are straightforward. Note the definition of `cons`, which underlies the finite set notation. The axiom of infinity gives us a set that contains 0 and is closed under successor (`succ`). Although this set is not uniquely defined, the theory names it (`Inf`) in order to simplify the construction of the natural numbers.

Further definitions appear in Fig. 3.5. Ordered pairs are defined in the standard way, $\langle a,b\rangle\equiv\{\{a\},\{a,b\}\}$. Recall that `Sigma`$(A,B)$ generalizes the Cartesian product of two sets. It is defined to be the union of all singleton sets $\{\langle x,y\rangle\}$, for $x\in A$ and $y\in B(x)$. This is a typical usage of general union.

```
Let_def           Let(s, f) == f(s)

Ball_def          Ball(A,P) == ALL x. x:A --> P(x)
Bex_def           Bex(A,P)  == EX x. x:A & P(x)

subset_def        A <= B  == ALL x:A. x:B
extension         A = B  <->  A <= B & B <= A

Union_iff         A : Union(C) <-> (EX B:C. A:B)
Pow_iff           A : Pow(B) <-> A <= B
foundation        A=0 | (EX x:A. ALL y:x. ~ y:A)

replacement       (ALL x:A. ALL y z. P(x,y) & P(x,z) --> y=z) ==>
                  b : PrimReplace(A,P) <-> (EX x:A. P(x,b))
```

The Zermelo-Fraenkel Axioms

```
Replace_def  Replace(A,P) ==
                  PrimReplace(A, %x y. (EX!z.P(x,z)) & P(x,y))
RepFun_def   RepFun(A,f)  == {y . x:A, y=f(x)}
the_def      The(P)       == Union({y . x:{0}, P(y)})
if_def       if(P,a,b)    == THE z. P & z=a | ~P & z=b
Collect_def  Collect(A,P) == {y . x:A, x=y & P(x)}
Upair_def    Upair(a,b)   ==
                {y. x:Pow(Pow(0)), (x=0 & y=a) | (x=Pow(0) & y=b)}
```

Consequences of replacement

```
Inter_def    Inter(A) == {x:Union(A) . ALL y:A. x:y}
Un_def       A Un  B  == Union(Upair(A,B))
Int_def      A Int B  == Inter(Upair(A,B))
Diff_def     A - B    == {x:A . x~:B}
```

Union, intersection, difference

Figure 3.4: Rules and axioms of ZF

```
cons_def     cons(a,A) == Upair(a,a) Un A
succ_def     succ(i) == cons(i,i)
infinity     0:Inf & (ALL y:Inf. succ(y): Inf)
```

Finite and infinite sets

```
Pair_def       <a,b>       == {{a,a}, {a,b}}
split_def      split(c,p) == THE y. EX a b. p=<a,b> & y=c(a,b)
fst_def        fst(A)      == split(%x y.x, p)
snd_def        snd(A)      == split(%x y.y, p)
Sigma_def      Sigma(A,B) == UN x:A. UN y:B(x). {<x,y>}
```

Ordered pairs and Cartesian products

```
converse_def   converse(r) == {z. w:r, EX x y. w=<x,y> & z=<y,x>}
domain_def     domain(r)   == {x. w:r, EX y. w=<x,y>}
range_def      range(r)    == domain(converse(r))
field_def      field(r)    == domain(r) Un range(r)
image_def      r `` A      == {y : range(r) . EX x:A. <x,y> : r}
vimage_def     r -`` A     == converse(r)``A
```

Operations on relations

```
lam_def     Lambda(A,b) == {<x,b(x)> . x:A}
apply_def   f`a         == THE y. <a,y> : f
Pi_def      Pi(A,B) == {f: Pow(Sigma(A,B)). ALL x:A. EX! y. <x,y>: f}
restrict_def   restrict(f,A) == lam x:A.f`x
```

Functions and general product

Figure 3.5: Further definitions of ZF

The projections `fst` and `snd` are defined in terms of the generalized projection `split`. The latter has been borrowed from Martin-Löf's Type Theory, and is often easier to use than `fst` and `snd`.

Operations on relations include converse, domain, range, and image. The set `Pi`(A, B) generalizes the space of functions between two sets. Note the simple definitions of λ-abstraction (using `RepFun`) and application (using a definite description). The function `restrict`(f, A) has the same values as f, but only over the domain A.

3.5 From basic lemmas to function spaces

Faced with so many definitions, it is essential to prove lemmas. Even trivial theorems like $A \cap B = B \cap A$ would be difficult to prove from the definitions alone. Isabelle's set theory derives many rules using a natural deduction style. Ideally, a natural deduction rule should introduce or eliminate just one operator, but this is not always practical. For most operators, we may forget its definition and use its derived rules instead.

3.5.1 Fundamental lemmas

Figure 3.6 presents the derived rules for the most basic operators. The rules for the bounded quantifiers resemble those for the ordinary quantifiers, but note that `ballE` uses a negated assumption in the style of Isabelle's classical reasoner. The congruence rules `ball_cong` and `bex_cong` are required by Isabelle's simplifier, but have few other uses. Congruence rules must be specially derived for all binding operators, and henceforth will not be shown.

Figure 3.6 also shows rules for the subset and equality relations (proof by extensionality), and rules about the empty set and the power set operator.

Figure 3.7 presents rules for replacement and separation. The rules for `Replace` and `RepFun` are much simpler than comparable rules for `PrimReplace` would be. The principle of separation is proved explicitly, although most proofs should use the natural deduction rules for `Collect`. The elimination rule `CollectE` is equivalent to the two destruction rules `CollectD1` and `CollectD2`, but each rule is suited to particular circumstances. Although too many rules can be confusing, there is no reason to aim for a minimal set of rules. See the file `ZF/ZF.ML` for a complete listing.

Figure 3.8 presents rules for general union and intersection. The empty intersection should be undefined. We cannot have $\bigcap(\emptyset) = V$ because V, the universal class, is not a set. All expressions denote something in `ZF` set theory; the definition of intersection implies $\bigcap(\emptyset) = \emptyset$, but this value is arbitrary. The rule `InterI` must have a premise to exclude the empty intersection. Some of the laws governing intersections require similar premises.

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

ball_cong   [| A=A';  !!x. x:A' ==> P(x) <-> P'(x) |] ==>
            (ALL x:A. P(x)) <-> (ALL x:A'. P'(x))

bexI        [| P(x);  x: A |] ==> EX x:A. P(x)
bexCI       [| ALL x:A. ~P(x) ==> P(a);  a: A |] ==> EX x:A.P(x)
bexE        [| EX x:A. P(x);  !!x. [| x:A; P(x) |] ==> Q |] ==> Q

bex_cong    [| A=A';  !!x. x:A' ==> P(x) <-> P'(x) |] ==>
            (EX x:A. P(x)) <-> (EX x:A'. P'(x))
```

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
```

Subsets and extensionality

```
emptyE          a:0 ==> P
empty_subsetI   0 <= A
equals0I        [| !!y. y:A ==> False |] ==> A=0
equals0D        [| A=0;  a:A |] ==> P

PowI            A <= B ==> A : Pow(B)
PowD            A : Pow(B)  ==>  A<=B
```

The empty set; power sets

Figure 3.6: Basic derived rules for ZF

```
ReplaceI      [| x: A;  P(x,b);  !!y. P(x,y) ==> y=b |] ==>
              b : {y. x:A, P(x,y)}

ReplaceE      [| b : {y. x:A, P(x,y)};
                 !!x. [| x: A;  P(x,b);  ALL y. P(x,y)-->y=b |] ==> R
              |] ==> R

RepFunI       [| a : A |] ==> f(a) : {f(x). x:A}
RepFunE       [| b : {f(x). x:A};
                 !!x.[| x:A;  b=f(x) |] ==> P |] ==> P

separation     a : {x:A. P(x)} <-> a:A & P(a)
CollectI       [| a:A;  P(a) |] ==> a : {x:A. P(x)}
CollectE       [| a : {x:A. P(x)};  [| a:A; P(a) |] ==> R |] ==> R
CollectD1      a : {x:A. P(x)} ==> a:A
CollectD2      a : {x:A. P(x)} ==> P(a)
```

Figure 3.7: Replacement and separation

```
UnionI    [| B: C;  A: B |] ==> A: Union(C)
UnionE    [| A : Union(C);  !!B.[| A: B;  B: C |] ==> R |] ==> R

InterI    [| !!x. x: C ==> A: x;  c:C |] ==> A : Inter(C)
InterD    [| A : Inter(C);  B : C |] ==> A : B
InterE    [| A : Inter(C);  A:B ==> R;  ~ B:C ==> R |] ==> R

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);  a: A |] ==> b: (INT x:A. B(x))
INT_E     [| b : (INT x:A. B(x));  a: A |] ==> b : B(a)
```

Figure 3.8: General union and intersection

```
pairing      a:Upair(b,c) <-> (a=b | a=c)
UpairI1      a : Upair(a,b)
UpairI2      b : Upair(a,b)
UpairE       [| a : Upair(b,c);  a = b ==> P;  a = c ==> P |] ==> P
```

Figure 3.9: Unordered pairs

```
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

DiffI        [| c : A;  ~ c : B |] ==> c : A - B
DiffD1       c : A - B ==> c : A
DiffD2       c : A - B ==> c ~: B
DiffE        [| c : A - B;  [| c:A; ~ c:B |] ==> P |] ==> P
```

Figure 3.10: Union, intersection, difference

```
consI1       a : cons(a,B)
consI2       a : B ==> a : cons(b,B)
consCI       (~ a:B ==> a=b) ==> a: cons(b,B)
consE        [| a : cons(b,A);  a=b ==> P;  a:A ==> P |] ==> P

singletonI   a : {a}
singletonE   [| a : {b}; a=b ==> P |] ==> P
```

Figure 3.11: Finite and singleton sets

```
succI1       i : succ(i)
succI2       i : j ==> i : succ(j)
succCI       (~ i:j ==> i=j) ==> i: succ(j)
succE        [| i : succ(j);  i=j ==> P;  i:j ==> P |] ==> P
succ_neq_0   [| succ(n)=0 |] ==> P
succ_inject  succ(m) = succ(n) ==> m=n
```

Figure 3.12: The successor function

```
the_equality     [| P(a);  !!x. P(x) ==> x=a |] ==> (THE x. P(x)) = a
theI             EX! x. P(x) ==> P(THE x. P(x))

if_P              P ==> if(P,a,b) = a
if_not_P         ~P ==> if(P,a,b) = b

mem_asym         [| a:b;  b:a |] ==> P
mem_irrefl       a:a ==> P
```

Figure 3.13: Descriptions; non-circularity

```
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    [| a:A;  !!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

Diff_subset       A-B <= A
Diff_contains     [| C<=A;  C Int B = 0 |] ==> C <= A-B

Collect_subset    Collect(A,P) <= A
```

Figure 3.14: Subset and lattice properties

3.5.2 Unordered pairs and finite sets

Figure 3.9 presents the principle of unordered pairing, along with its derived rules. Binary union and intersection are defined in terms of ordered pairs (Fig. 3.10). Set difference is also included. The rule `UnCI` is useful for classical reasoning about unions, like `disjCI`; it supersedes `UnI1` and `UnI2`, but these rules are often easier to work with. For intersection and difference we have both elimination and destruction rules. Again, there is no reason to provide a minimal rule set.

Figure 3.11 is concerned with finite sets: it presents rules for `cons`, the finite set constructor, and rules for singleton sets. Figure 3.12 presents derived rules for the successor function, which is defined in terms of `cons`. The proof that `succ` is injective appears to require the Axiom of Foundation.

Definite descriptions (`THE`) are defined in terms of the singleton set $\{0\}$, but their derived rules fortunately hide this (Fig. 3.13). The rule `theI` is difficult to apply because of the two occurrences of $?P$. However, `the_equality` does not have this problem and the files contain many examples of its use.

Finally, the impossibility of having both $a \in b$ and $b \in a$ (`mem_asym`) is proved by applying the Axiom of Foundation to the set $\{a, b\}$. The impossibility of $a \in a$ is a trivial consequence.

See the file `ZF/upair.ML` for full proofs of the rules discussed in this section.

3.5.3 Subset and lattice properties

The subset relation is a complete lattice. Unions form least upper bounds; non-empty intersections form greatest lower bounds. Figure 3.14 shows the corre-

```
Pair_inject1    <a,b> = <c,d> ==> a=c
Pair_inject2    <a,b> = <c,d> ==> b=d
Pair_inject     [| <a,b> = <c,d>;  [| a=c; b=d |] ==> P |] ==> P
Pair_neq_0      <a,b>=0 ==> P

fst_conv        fst(<a,b>) = a
snd_conv        snd(<a,b>) = b
split           split(%x y.c(x,y), <a,b>) = c(a,b)

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

SigmaE2         [| <a,b> : Sigma(A,B);
                   [| a:A;  b:B(a) |] ==> P   |] ==> P
```

Figure 3.15: Ordered pairs; projections; general sums

sponding rules. A few other laws involving subsets are included. Proofs are in the file `ZF/subset.ML`.

Reasoning directly about subsets often yields clearer proofs than reasoning about the membership relation. Section 3.9 below presents an example of this, proving the equation $\texttt{Pow}(A) \cap \texttt{Pow}(B) = \texttt{Pow}(A \cap B)$.

3.5.4 Ordered pairs

Figure 3.15 presents the rules governing ordered pairs, projections and general sums. File `ZF/pair.ML` contains the full (and tedious) proof that $\{\{a\}, \{a, b\}\}$ functions as an ordered pair. This property is expressed as two destruction rules, `Pair_inject1` and `Pair_inject2`, and equivalently as the elimination rule `Pair_inject`.

The rule `Pair_neq_0` asserts $\langle a, b\rangle \neq \emptyset$. This is a property of $\{\{a\}, \{a, b\}\}$, and need not hold for other encodings of ordered pairs. The non-standard ordered pairs mentioned below satisfy $\langle\emptyset; \emptyset\rangle = \emptyset$.

The natural deduction rules `SigmaI` and `SigmaE` assert that $\texttt{Sigma}(A, B)$ consists of all pairs of the form $\langle x, y\rangle$, for $x \in A$ and $y \in B(x)$. The rule `SigmaE2` merely states that $\langle a, b\rangle \in \texttt{Sigma}(A, B)$ implies $a \in A$ and $b \in B(a)$.

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 translated recursively: $\texttt{\%<}x\texttt{,}y\texttt{,}z\texttt{>.}t \leadsto \texttt{\%<}x\texttt{,<}y\texttt{,}z\texttt{>>.}t \leadsto \texttt{split(\%}x\texttt{.\%<}y\texttt{,}z\texttt{>.}t\texttt{)} \leadsto \texttt{split(\%}x\texttt{.split(\%}y\ z\texttt{.}t\texttt{))}$. The reverse translation is performed upon printing.

```
domainI        <a,b>: r ==> a : domain(r)
domainE        [| a : domain(r);  !!y. <a,y>: r ==> P |] ==> P
domain_subset  domain(Sigma(A,B)) <= A

rangeI         <a,b>: r ==> b : range(r)
rangeE         [| b : range(r);  !!x. <x,b>: r ==> P |] ==> P
range_subset   range(A*B) <= B

fieldI1        <a,b>: r ==> a : field(r)
fieldI2        <a,b>: r ==> b : field(r)
fieldCI        (~ <c,a>:r ==> <a,b>: r) ==> a : field(r)

fieldE         [| a : field(r);
                  !!x. <a,x>: r ==> P;
                  !!x. <x,a>: r ==> P
               |] ==> P

field_subset   field(A*A) <= A
```

Figure 3.16: Domain, range and field of a relation

```
imageI         [| <a,b>: r;  a:A |] ==> b : r``A
imageE         [| b: r``A;  !!x.[| <x,b>: r;  x:A |] ==> P |] ==> P

vimageI        [| <a,b>: r;  b:B |] ==> a : r-``B
vimageE        [| a: r-``B;  !!x.[| <a,x>: r;  x:B |] ==> P |] ==> P
```

Figure 3.17: Image and inverse image

! 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 affects proofs. For example the term `(%<x,y>.<y,x>)<a,b>` requires the theorem `split` 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*

Choice: `THE` *pattern* `.` *P*

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

Comprehension: `{` *pattern*`:`*A* `.` *P* `}`

```
fun_is_rel      f: Pi(A,B) ==> f <= Sigma(A,B)

apply_equality  [| <a,b>: f;  f: Pi(A,B) |] ==> f`a = b
apply_equality2 [| <a,b>: f;  <a,c>: f;  f: Pi(A,B) |] ==> b=c

apply_type      [| f: Pi(A,B);  a:A |] ==> f`a : B(a)
apply_Pair      [| f: Pi(A,B);  a:A |] ==> <a,f`a>: f
apply_iff       f: Pi(A,B) ==> <a,b>: f <-> a:A & f`a = b

fun_extension   [| f : Pi(A,B);  g: Pi(A,D);
                   !!x. x:A ==> f`x = g`x     |] ==> f=g

domain_type     [| <a,b> : f;  f: Pi(A,B) |] ==> a : A
range_type      [| <a,b> : f;  f: Pi(A,B) |] ==> b : B(a)

Pi_type         [| f: A->C;  !!x. x:A ==> f`x: B(x) |] ==> f: Pi(A,B)
domain_of_fun   f: Pi(A,B) ==> domain(f)=A
range_of_fun    f: Pi(A,B) ==> f: A->range(f)

restrict        a : A ==> restrict(f,A) ` a = f`a
restrict_type   [| !!x. x:A ==> f`x: B(x) |] ==>
                restrict(f,A) : Pi(A,B)
```

Figure 3.18: Functions

```
lamI         a:A ==> <a,b(a)> : (lam x:A. b(x))
lamE         [| p: (lam x:A. b(x));  !!x.[| x:A; p=<x,b(x)> |] ==> P
             |] ==>  P

lam_type     [| !!x. x:A ==> b(x): B(x) |] ==> (lam x:A.b(x)) : Pi(A,B)

beta         a : A ==> (lam x:A.b(x)) ` a = b(a)
eta          f : Pi(A,B) ==> (lam x:A. f`x) = f
```

Figure 3.19: λ-abstraction

3.5.5 Relations

Figure 3.16 presents rules involving relations, which are sets of ordered pairs. The converse of a relation r is the set of all pairs $\langle y, x\rangle$ such that $\langle x, y\rangle \in r$; if r is a function, then `converse`(r) is its inverse. The rules for the domain operation, namely `domainI` and `domainE`, assert that `domain`(r) consists of all x such that r contains some pair of the form $\langle x, y\rangle$. The range operation is similar, and the field of a relation is merely the union of its domain and range.

Figure 3.17 presents rules for images and inverse images. Note that these operations are generalisations of range and domain, respectively. See the file `ZF/domrange.ML` for derivations of the rules.

```
fun_empty            0: 0->0
fun_single           {<a,b>} : {a} -> {b}

fun_disjoint_Un      [| f: A->B;  g: C->D;  A Int C = 0  |] ==>
                     (f Un g) : (A Un C) -> (B Un D)

fun_disjoint_apply1  [| a:A;  f: A->B;  g: C->D;  A Int C = 0 |] ==>
                     (f Un g)`a = f`a

fun_disjoint_apply2  [| c:C;  f: A->B;  g: C->D;  A Int C = 0 |] ==>
                     (f Un g)`c = g`c
```

Figure 3.20: Constructing functions from smaller sets

3.5.6 Functions

Functions, represented by graphs, are notoriously difficult to reason about. The file `ZF/func.ML` derives many rules, which overlap more than they ought. This section presents the more important rules.

Figure 3.18 presents the basic properties of `Pi`(A, B), the generalized function space. For example, if f is a function and $\langle a, b\rangle \in f$, then $f`a = b$ (`apply_equality`). Two functions are equal provided they have equal domains and deliver equals results (`fun_extension`).

By `Pi_type`, a function typing of the form $f \in A \to C$ can be refined to the dependent typing $f \in \prod_{x\in A} B(x)$, given a suitable family of sets $\{B(x)\}_{x\in A}$. Conversely, by `range_of_fun`, any dependent typing can be flattened to yield a function type of the form $A \to C$; here, $C = \texttt{range}(f)$.

Among the laws for λ-abstraction, `lamI` and `lamE` describe the graph of the generated function, while `beta` and `eta` are the standard conversions. We essentially have a dependently-typed λ-calculus (Fig. 3.19).

Figure 3.20 presents some rules that can be used to construct functions explicitly. We start with functions consisting of at most one pair, and may form the union of two functions provided their domains are disjoint.

3.6 Further developments

The next group of developments is complex and extensive, and only highlights can be covered here. It involves many theories and ML files of proofs.

Figure 3.21 presents commutative, associative, distributive, and idempotency laws of union and intersection, along with other equations. See file `ZF/equalities.ML`.

Theory `Bool` defines $\{0, 1\}$ as a set of booleans, with the usual operators including a conditional (Fig. 3.22). Although `ZF` is a first-order theory, you can obtain the effect of higher-order logic using `bool`-valued functions, for example.

```
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)

Diff_cancel        A-A = 0
Diff_disjoint      A Int (B-A) = 0
Diff_partition     A<=B ==> A Un (B-A) = B
double_complement  [| A<=B; B<= C |] ==> (B - (C-A)) = A
Diff_Un            A - (B Un C) = (A-B) Int (A-C)
Diff_Int           A - (B Int C) = (A-B) Un (A-C)

Union_Un_distrib   Union(A Un B) = Union(A) Un Union(B)
Inter_Un_distrib   [| a:A;  b:B |] ==>
                   Inter(A Un B) = Inter(A) Int Inter(B)

Int_Union_RepFun   A Int Union(B) = (UN C:B. A Int C)

Un_Inter_RepFun    b:B ==>
                   A Un Inter(B) = (INT C:B. A Un C)

SUM_Un_distrib1    (SUM x:A Un B. C(x)) =
                   (SUM x:A. C(x)) Un (SUM x:B. C(x))

SUM_Un_distrib2    (SUM x:C. A(x) Un B(x)) =
                   (SUM x:C. A(x))  Un  (SUM x:C. B(x))

SUM_Int_distrib1   (SUM x:A Int B. C(x)) =
                   (SUM x:A. C(x)) Int (SUM x:B. C(x))

SUM_Int_distrib2   (SUM x:C. A(x) Int B(x)) =
                   (SUM x:C. A(x)) Int (SUM x:C. B(x))
```

Figure 3.21: Equalities

```
bool_def      bool == {0,1}
cond_def      cond(b,c,d) == if(b=1,c,d)
not_def       not(b)  == cond(b,0,1)
and_def       a and b == cond(a,b,0)
or_def        a or b  == cond(a,1,b)
xor_def       a xor b == cond(a,not(b),b)

bool_1I       1 : bool
bool_0I       0 : bool
boolE         [| c: bool;  c=1 ==> P;  c=0 ==> P |] ==> P
cond_1        cond(1,c,d) = c
cond_0        cond(0,c,d) = d
```

Figure 3.22: The booleans

The constant `1` is translated to `succ(0)`.

Theory `Sum` defines the disjoint union of two sets, with injections and a case analysis operator (Fig. 3.23). Disjoint unions play a role in datatype definitions, particularly when there is mutual recursion [35].

Theory `QPair` defines a notion of ordered pair that admits non-well-founded tupling (Fig. 3.24). Such pairs are written `<`a`;`b`>`. It also defines the eliminator `qsplit`, the converse operator `qconverse`, and the summation operator `QSigma`. These are completely analogous to the corresponding versions for standard ordered pairs. The theory goes on to define a non-standard notion of disjoint sum using non-standard pairs. All of these concepts satisfy the same properties as their standard counterparts; in addition, `<`a`;`b`>` is continuous. The theory supports coinductive definitions, for example of infinite lists [40].

The Knaster-Tarski Theorem states that every monotone function over a complete lattice has a fixedpoint. Theory `Fixedpt` proves the Theorem only for a particular lattice, namely the lattice of subsets of a set (Fig. 3.25). The theory defines least and greatest fixedpoint operators with corresponding induction and coinduction rules. These are essential to many definitions that follow, including the natural numbers and the transitive closure operator. The (co)inductive definition package also uses the fixedpoint operators [34]. See Davey and Priestley [9] for more on the Knaster-Tarski Theorem and my paper [35] for discussion of the Isabelle proofs.

Monotonicity properties are proved for most of the set-forming operations: union, intersection, Cartesian product, image, domain, range, etc. These are useful for applying the Knaster-Tarski Fixedpoint Theorem. The proofs themselves are trivial applications of Isabelle's classical reasoner. See file `ZF/mono.ML`.

The theory `Perm` is concerned with permutations (bijections) and related concepts. These include composition of relations, the identity relation, and three specialized function spaces: injective, surjective and bijective. Figure 3.26 displays many of their properties that have been proved. These results are fundamental

symbol	*meta-type*	*priority*	*description*
`+`	$[i, i] \Rightarrow i$	Right 65	disjoint union operator
`Inl Inr`	$i \Rightarrow i$		injections
`case`	$[i \Rightarrow i, i \Rightarrow i, i] \Rightarrow i$		conditional for $A + B$

```
sum_def         A+B == {0}*A Un {1}*B
Inl_def         Inl(a) == <0,a>
Inr_def         Inr(b) == <1,b>
case_def        case(c,d,u) == split(%y z. cond(y, d(z), c(z)), u)

sum_InlI        a : A ==> Inl(a) : A+B
sum_InrI        b : B ==> Inr(b) : A+B

Inl_inject      Inl(a)=Inl(b) ==> a=b
Inr_inject      Inr(a)=Inr(b) ==> a=b
Inl_neq_Inr     Inl(a)=Inr(b) ==> P

sumE2   u: A+B ==> (EX x. x:A & u=Inl(x)) | (EX y. y:B & u=Inr(y))

case_Inl        case(c,d,Inl(a)) = c(a)
case_Inr        case(c,d,Inr(b)) = d(b)
```

Figure 3.23: Disjoint unions

```
QPair_def       <a;b> == a+b
qsplit_def      qsplit(c,p)  == THE y. EX a b. p=<a;b> & y=c(a,b)
qfsplit_def     qfsplit(R,z) == EX x y. z=<x;y> & R(x,y)
qconverse_def   qconverse(r) == {z. w:r, EX x y. w=<x;y> & z=<y;x>}
QSigma_def      QSigma(A,B)  == UN x:A. UN y:B(x). {<x;y>}

qsum_def        A <+> B      == ({0} <*> A) Un ({1} <*> B)
QInl_def        QInl(a)      == <0;a>
QInr_def        QInr(b)      == <1;b>
qcase_def       qcase(c,d)   == qsplit(%y z. cond(y, d(z), c(z)))
```

Figure 3.24: Non-standard pairs, products and sums

```
bnd_mono_def   bnd_mono(D,h) ==
                 h(D)<=D & (ALL W X. W<=X --> X<=D --> h(W) <= h(X))

lfp_def        lfp(D,h) == Inter({X: Pow(D). h(X) <= X})
gfp_def        gfp(D,h) == Union({X: Pow(D). X <= h(X)})


lfp_lowerbound [| h(A) <= A;  A<=D |] ==> lfp(D,h) <= A

lfp_subset     lfp(D,h) <= D

lfp_greatest   [| bnd_mono(D,h);
                  !!X. [| h(X) <= X;  X<=D |] ==> A<=X
               |] ==> A <= lfp(D,h)

lfp_Tarski     bnd_mono(D,h) ==> lfp(D,h) = h(lfp(D,h))

induct         [| a : lfp(D,h);  bnd_mono(D,h);
                  !!x. x : h(Collect(lfp(D,h),P)) ==> P(x)
               |] ==> P(a)

lfp_mono       [| bnd_mono(D,h);  bnd_mono(E,i);
                  !!X. X<=D ==> h(X) <= i(X)
               |] ==> lfp(D,h) <= lfp(E,i)

gfp_upperbound [| A <= h(A);  A<=D |] ==> A <= gfp(D,h)

gfp_subset     gfp(D,h) <= D

gfp_least      [| bnd_mono(D,h);
                  !!X. [| X <= h(X);  X<=D |] ==> X<=A
               |] ==> gfp(D,h) <= A

gfp_Tarski     bnd_mono(D,h) ==> gfp(D,h) = h(gfp(D,h))

coinduct       [| bnd_mono(D,h); a: X; X <= h(X Un gfp(D,h)); X <= D
               |] ==> a : gfp(D,h)

gfp_mono       [| bnd_mono(D,h);  D <= E;
                  !!X. X<=D ==> h(X) <= i(X)
               |] ==> gfp(D,h) <= gfp(E,i)
```

Figure 3.25: Least and greatest fixedpoints

symbol	*meta-type*	*priority*	*description*
`O`	$[i,i] \Rightarrow i$	Right 60	composition ($\circ$)
`id`	$i \Rightarrow i$		identity function
`inj`	$[i,i] \Rightarrow i$		injective function space
`surj`	$[i,i] \Rightarrow i$		surjective function space
`bij`	$[i,i] \Rightarrow i$		bijective function space

```
comp_def  r O s     == {xz : domain(s)*range(r) .
                        EX x y z. xz=<x,z> & <x,y>:s & <y,z>:r}
id_def    id(A)     == (lam x:A. x)
inj_def   inj(A,B)  == { f: A->B. ALL w:A. ALL x:A. f`w=f`x --> w=x }
surj_def  surj(A,B) == { f: A->B . ALL y:B. EX x:A. f`x=y }
bij_def   bij(A,B)  == inj(A,B) Int surj(A,B)


left_inverse     [| f: inj(A,B);  a: A |] ==> converse(f)`(f`a) = a
right_inverse    [| f: inj(A,B);  b: range(f) |] ==>
                 f`(converse(f)`b) = b

inj_converse_inj f: inj(A,B) ==> converse(f): inj(range(f), A)
bij_converse_bij f: bij(A,B) ==> converse(f): bij(B,A)

comp_type        [| s<=A*B;  r<=B*C |] ==> (r O s) <= A*C
comp_assoc       (r O s) O t = r O (s O t)

left_comp_id     r<=A*B ==> id(B) O r = r
right_comp_id    r<=A*B ==> r O id(A) = r

comp_func        [| g:A->B; f:B->C |] ==> (f O g):A->C
comp_func_apply  [| g:A->B; f:B->C; a:A |] ==> (f O g)`a = f`(g`a)

comp_inj         [| g:inj(A,B);  f:inj(B,C)  |] ==> (f O g):inj(A,C)
comp_surj        [| g:surj(A,B); f:surj(B,C) |] ==> (f O g):surj(A,C)
comp_bij         [| g:bij(A,B); f:bij(B,C) |] ==> (f O g):bij(A,C)

left_comp_inverse     f: inj(A,B) ==> converse(f) O f = id(A)
right_comp_inverse    f: surj(A,B) ==> f O converse(f) = id(B)

bij_disjoint_Un
    [| f: bij(A,B);  g: bij(C,D);  A Int C = 0;  B Int D = 0 |] ==>
    (f Un g) : bij(A Un C, B Un D)

restrict_bij  [| f:inj(A,B);  C<=A |] ==> restrict(f,C): bij(C, f``C)
```

Figure 3.26: Permutations

to a treatment of equipollence and cardinality.

Theory `Nat` defines the natural numbers and mathematical induction, along with a case analysis operator. The set of natural numbers, here called `nat`, is known in set theory as the ordinal ω.

Theory `Arith` defines primitive recursion and goes on to develop arithmetic on the natural numbers (Fig. 3.27). It defines addition, multiplication, subtraction, division, and remainder. Many of their properties are proved: commutative, associative and distributive laws, identity and cancellation laws, etc. The most interesting result is perhaps the theorem $a \bmod b + (a/b) \times b = a$. Division and remainder are defined by repeated subtraction, which requires well-founded rather than primitive recursion; the termination argument relies on the divisor's being non-zero.

Theory `Univ` defines a 'universe' $\texttt{univ}(A)$, for constructing datatypes such as trees. This set contains A and the natural numbers. Vitally, it is closed under finite products: $\texttt{univ}(A) \times \texttt{univ}(A) \subseteq \texttt{univ}(A)$. This theory also defines the cumulative hierarchy of axiomatic set theory, which traditionally is written V_α for an ordinal α. The 'universe' is a simple generalization of V_ω.

Theory `QUniv` defines a 'universe' $\texttt{quniv}(A)$, for constructing codatatypes such as streams. It is analogous to $\texttt{univ}(A)$ (and is defined in terms of it) but is closed under the non-standard product and sum.

Theory `Finite` (Figure 3.28) defines the finite set operator; $\texttt{Fin}(A)$ is the set of all finite sets over A. The theory employs Isabelle's inductive definition package, which proves various rules automatically. The induction rule shown is stronger than the one proved by the package. The theory also defines the set of all finite functions between two given sets.

Figure 3.29 presents the set of lists over A, $\texttt{list}(A)$. The definition employs Isabelle's datatype package, which defines the introduction and induction rules automatically, as well as the constructors and case operator (`list_case`). See file `ZF/List.ML`. The file `ZF/ListFn.thy` proceeds to define structural recursion and the usual list functions.

The constructions of the natural numbers and lists make use of a suite of operators for handling recursive function definitions. I have described the developments in detail elsewhere [35]. Here is a brief summary:

- Theory `Trancl` defines the transitive closure of a relation (as a least fixedpoint).

- Theory `WF` proves the Well-Founded Recursion Theorem, using an elegant approach of Tobias Nipkow. This theorem permits general recursive definitions within set theory.

- Theory `Ord` defines the notions of transitive set and ordinal number. It derives transfinite induction. A key definition is **less than**: $i < j$ if and

symbol	*meta-type*	*priority*	*description*
`nat`	i		set of natural numbers
`nat_case`	$[i, i \Rightarrow i, i] \Rightarrow i$		conditional for *nat*
`rec`	$[i, i, [i, i] \Rightarrow i] \Rightarrow i$		recursor for *nat*
`#*`	$[i, i] \Rightarrow i$	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

```
nat_def       nat == lfp(lam r: Pow(Inf). {0} Un {succ(x). x:r}

nat_case_def  nat_case(a,b,k) ==
              THE y. k=0 & y=a | (EX x. k=succ(x) & y=b(x))

rec_def       rec(k,a,b) ==
              transrec(k, %n f. nat_case(a, %m. b(m, f`m), n))

add_def       m#+n    == rec(m, n, %u v.succ(v))
diff_def      m#-n    == rec(n, m, %u v. rec(v, 0, %x y.x))
mult_def      m#*n    == rec(m, 0, %u v. n #+ v)
mod_def       m mod n == transrec(m, %j f. if(j:n, j, f`(j#-n)))
div_def       m div n == transrec(m, %j f. if(j:n, 0, succ(f`(j#-n))))


nat_0I        0 : nat
nat_succI     n : nat ==> succ(n) : nat

nat_induct
    [| n: nat;  P(0);  !!x. [| x: nat;  P(x) |] ==> P(succ(x))
    |] ==> P(n)

nat_case_0    nat_case(a,b,0) = a
nat_case_succ nat_case(a,b,succ(m)) = b(m)

rec_0         rec(0,a,b) = a
rec_succ      rec(succ(m),a,b) = b(m, rec(m,a,b))

mult_type     [| m:nat;  n:nat |] ==> m #* n : nat
mult_0        0 #* n = 0
mult_succ     succ(m) #* n = n #+ (m #* n)
mult_commute  [| m:nat; n:nat |] ==> m #* n = n #* m
add_mult_dist [| m:nat; k:nat |] ==> (m #+ n) #* k = (m #* k)#+(n #* k)
mult_assoc
    [| m:nat;  n:nat;  k:nat |] ==> (m #* n) #* k = m #* (n #* k)
mod_quo_equality
    [| 0:n;  m:nat;  n:nat |] ==> (m div n)#*n #+ m mod n = m
```

Figure 3.27: The natural numbers

```
Fin.emptyI      0 : Fin(A)
Fin.consI       [| a: A;  b: Fin(A) |] ==> cons(a,b) : Fin(A)

Fin_induct
    [| b: Fin(A);
       P(0);
       !!x y. [| x: A;  y: Fin(A);  x~:y;  P(y) |] ==> P(cons(x,y))
    |] ==> P(b)

Fin_mono        A<=B ==> Fin(A) <= Fin(B)
Fin_UnI         [| b: Fin(A);  c: Fin(A) |] ==> b Un c : Fin(A)
Fin_UnionI      C : Fin(Fin(A)) ==> Union(C) : Fin(A)
Fin_subset      [| c<=b;  b: Fin(A) |] ==> c: Fin(A)
```

Figure 3.28: The finite set operator

only if i and j are both ordinals and $i \in j$. As a special case, it includes less than on the natural numbers.

- Theory `Epsilon` derives ε-induction and ε-recursion, which are generalisations of transfinite induction and recursion. It also defines `rank`(x), which is the least ordinal α such that x is constructed at stage α of the cumulative hierarchy (thus $x \in V_{\alpha+1}$).

Other important theories lead to a theory of cardinal numbers. They have not yet been written up anywhere. Here is a summary:

- Theory `Rel` defines the basic properties of relations, such as (ir)reflexivity, (a)symmetry, and transitivity.
- Theory `EquivClass` develops a theory of equivalence classes, not using the Axiom of Choice.
- Theory `Order` defines partial orderings, total orderings and wellorderings.
- Theory `OrderArith` defines orderings on sum and product sets. These can be used to define ordinal arithmetic and have applications to cardinal arithmetic.
- Theory `OrderType` defines order types. Every wellordering is equivalent to a unique ordinal, which is its order type.
- Theory `Cardinal` defines equipollence and cardinal numbers.
- Theory `CardinalArith` defines cardinal addition and multiplication, and proves their elementary laws. It proves that there is no greatest cardinal. It also proves a deep result, namely $\kappa \otimes \kappa = \kappa$ for every infinite cardinal κ; see Kunen [19, page 29]. None of these results assume the Axiom of Choice, which complicates their proofs considerably.

`list`	$i \Rightarrow i$		lists over some set
`list_case`	$[i, [i, i] \Rightarrow i, i] \Rightarrow i$		conditional for $\mathit{list}(A)$
`list_rec`	$[i, i, [i, i, i] \Rightarrow i] \Rightarrow i$		recursor for $\mathit{list}(A)$
`map`	$[i \Rightarrow i, i] \Rightarrow i$		mapping functional
`length`	$i \Rightarrow i$		length of a list
`rev`	$i \Rightarrow i$		reverse of a list
`@`	$[i, i] \Rightarrow i$	Right 60	append for lists
`flat`	$i \Rightarrow i$		append of list of lists

```
list_rec_def    list_rec(l,c,h) ==
                Vrec(l, %l g.list_case(c, %x xs. h(x, xs, g`xs), l))

map_def         map(f,l)  == list_rec(l,  0,  %x xs r. <f(x), r>)
length_def      length(l) == list_rec(l,  0,  %x xs r. succ(r))
app_def         xs@ys     == list_rec(xs, ys, %x xs r. <x,r>)
rev_def         rev(l)    == list_rec(l,  0,  %x xs r. r @ <x,0>)
flat_def        flat(ls)  == list_rec(ls, 0,  %l ls r. l @ r)


NilI            Nil : list(A)
ConsI           [| a: A;  l: list(A) |] ==> Cons(a,l) : list(A)

List.induct
    [| l: list(A);
       P(Nil);
       !!x y. [| x: A;  y: list(A);  P(y) |] ==> P(Cons(x,y))
    |] ==> P(l)

Cons_iff        Cons(a,l)=Cons(a',l') <-> a=a' & l=l'
Nil_Cons_iff    ~ Nil=Cons(a,l)

list_mono       A<=B ==> list(A) <= list(B)

list_rec_Nil    list_rec(Nil,c,h) = c
list_rec_Cons   list_rec(Cons(a,l), c, h) = h(a, l, list_rec(l,c,h))

map_ident       l: list(A) ==> map(%u.u, l) = l
map_compose     l: list(A) ==> map(h, map(j,l)) = map(%u.h(j(u)), l)
map_app_distrib xs: list(A) ==> map(h, xs@ys) = map(h,xs) @ map(h,ys)
map_type
    [| l: list(A);  !!x. x: A ==> h(x): B |] ==> map(h,l) : list(B)
map_flat
    ls: list(list(A)) ==> map(h, flat(ls)) = flat(map(map(h),ls))
```

Figure 3.29: Lists

The following developments involve the Axiom of Choice (AC):

- Theory `AC` asserts the Axiom of Choice and proves some simple equivalent forms.
- Theory `Zorn` proves Hausdorff's Maximal Principle, Zorn's Lemma and the Wellordering Theorem, following Abrial and Laffitte [1].
- Theory `Cardinal_AC` uses AC to prove simplified theorems about the cardinals. It also proves a theorem needed to justify infinitely branching datatype declarations: if κ is an infinite cardinal and $|X(\alpha)| \leq \kappa$ for all $\alpha < \kappa$ then $|\bigcup_{\alpha<\kappa} X(\alpha)| \leq \kappa$.
- Theory `InfDatatype` proves theorems to justify infinitely branching datatypes. Arbitrary index sets are allowed, provided their cardinalities have an upper bound. The theory also justifies some unusual cases of finite branching, involving the finite powerset operator and the finite function space operator.

3.7 Simplification rules

`ZF` does not merely inherit simplification from `FOL`, but modifies it extensively. File `ZF/simpdata.ML` contains the details.

The extraction of rewrite rules takes set theory primitives into account. It can strip bounded universal quantifiers from a formula; for example, $\forall x \in A . f(x) = g(x)$ yields the conditional rewrite rule $x \in A \Longrightarrow f(x) = g(x)$. Given $a \in \{x \in A . P(x)\}$ it extracts rewrite rules from $a \in A$ and $P(a)$. It can also break down $a \in A \cap B$ and $a \in A - B$.

The default simplification set contains congruence rules for all the binding operators of `ZF`. It contains all the conversion rules, such as `fst` and `snd`, as well as the rewrites shown in Fig. 3.30. See the file `ZF/simpdata.ML` for a fuller list.

3.8 The examples directories

Directory `HOL/IMP` contains a mechanised version of a semantic equivalence proof taken from Winskel [50]. It formalises the denotational and operational semantics of a simple while-language, then proves the two equivalent. It contains several datatype and inductive definitions, and demonstrates their use.

The directory `ZF/ex` contains further developments in `ZF` set theory. Here is an overview; see the files themselves for more details. I describe much of this material in other publications [33, 35, 34].

$$
\begin{array}{rcl}
a \in \emptyset & \leftrightarrow & \bot \\
a \in A \cup B & \leftrightarrow & a \in A \lor a \in B \\
a \in A \cap B & \leftrightarrow & a \in A \land a \in B \\
a \in A - B & \leftrightarrow & a \in A \land \neg(a \in B) \\
\langle a, b\rangle \in \texttt{Sigma}(A, B) & \leftrightarrow & a \in A \land b \in B(a) \\
a \in \texttt{Collect}(A, P) & \leftrightarrow & a \in A \land P(a) \\
(\forall x \in \emptyset \,.\, P(x)) & \leftrightarrow & \top \\
(\forall x \in A \,.\, \top) & \leftrightarrow & \top
\end{array}
$$

Figure 3.30: Some rewrite rules for set theory

- File `misc.ML` contains miscellaneous examples such as Cantor's Theorem, the Schröder-Bernstein Theorem and the 'Composition of homomorphisms' challenge [4].
- Theory `Ramsey` proves the finite exponent 2 version of Ramsey's Theorem, following Basin and Kaufmann's presentation [3].
- Theory `Integ` develops a theory of the integers as equivalence classes of pairs of natural numbers.
- Theory `Primrec` develops some computation theory. It inductively defines the set of primitive recursive functions and presents a proof that Ackermann's function is not primitive recursive.
- Theory `Primes` defines the Greatest Common Divisor of two natural numbers and and the "divides" relation.
- Theory `Bin` defines a datatype for two's complement binary integers, then proves rewrite rules to perform binary arithmetic. For instance, $1359 \times -2468 = -3354012$ takes under 14 seconds.
- Theory `BT` defines the recursive data structure $\texttt{bt}(A)$, labelled binary trees.
- Theory `Term` defines a recursive data structure for terms and term lists. These are simply finite branching trees.
- Theory `TF` defines primitives for solving mutually recursive equations over sets. It constructs sets of trees and forests as an example, including induction and recursion rules that handle the mutual recursion.

- Theory `Prop` proves soundness and completeness of propositional logic [35]. This illustrates datatype definitions, inductive definitions, structural induction and rule induction.
- Theory `ListN` inductively defines the lists of n elements [30].
- Theory `Acc` inductively defines the accessible part of a relation [30].
- Theory `Comb` defines the datatype of combinators and inductively defines contraction and parallel contraction. It goes on to prove the Church-Rosser Theorem. This case study follows Camilleri and Melham [5].
- Theory `LList` defines lazy lists and a coinduction principle for proving equations between them.

3.9 A proof about powersets

To demonstrate high-level reasoning about subsets, let us prove the equation $\texttt{Pow}(A) \cap \texttt{Pow}(B) = \texttt{Pow}(A \cap B)$. Compared with first-order logic, set theory involves a maze of rules, and theorems have many different proofs. Attempting other proofs of the theorem might be instructive. This proof exploits the lattice properties of intersection. It also uses the monotonicity of the powerset operation, from `ZF/mono.ML`:

```
Pow_mono      A<=B ==> Pow(A) <= Pow(B)
```

We enter the goal and make the first step, which breaks the equation into two inclusions by extensionality:

```
goal thy "Pow(A Int B) = Pow(A) Int Pow(B)";
  Level 0
  Pow(A Int B) = Pow(A) Int Pow(B)
   1. Pow(A Int B) = Pow(A) Int Pow(B)
by (resolve_tac [equalityI] 1);
  Level 1
  Pow(A Int B) = Pow(A) Int Pow(B)
   1. Pow(A Int B) <= Pow(A) Int Pow(B)
   2. Pow(A) Int Pow(B) <= Pow(A Int B)
```

Both inclusions could be tackled straightforwardly using `subsetI`. A shorter proof results from noting that intersection forms the greatest lower bound:

```
by (resolve_tac [Int_greatest] 1);
  Level 2
  Pow(A Int B) = Pow(A) Int Pow(B)
   1. Pow(A Int B) <= Pow(A)
   2. Pow(A Int B) <= Pow(B)
   3. Pow(A) Int Pow(B) <= Pow(A Int B)
```

Subgoal 1 follows by applying the monotonicity of `Pow` to $A \cap B \subseteq A$; subgoal 2 follows similarly:

```
by (resolve_tac [Int_lower1 RS Pow_mono] 1);
  Level 3
  Pow(A Int B) = Pow(A) Int Pow(B)
   1. Pow(A Int B) <= Pow(B)
   2. Pow(A) Int Pow(B) <= Pow(A Int B)
by (resolve_tac [Int_lower2 RS Pow_mono] 1);
  Level 4
  Pow(A Int B) = Pow(A) Int Pow(B)
   1. Pow(A) Int Pow(B) <= Pow(A Int B)
```

We are left with the opposite inclusion, which we tackle in the straightforward way:

```
by (resolve_tac [subsetI] 1);
  Level 5
  Pow(A Int B) = Pow(A) Int Pow(B)
   1. !!x. x : Pow(A) Int Pow(B) ==> x : Pow(A Int B)
```

The subgoal is to show $x \in \texttt{Pow}(A \cap B)$ assuming $x \in \texttt{Pow}(A) \cap \texttt{Pow}(B)$; eliminating this assumption produces two subgoals. The rule `IntE` treats the intersection like a conjunction instead of unfolding its definition.

```
by (eresolve_tac [IntE] 1);
  Level 6
  Pow(A Int B) = Pow(A) Int Pow(B)
   1. !!x. [| x : Pow(A); x : Pow(B) |] ==> x : Pow(A Int B)
```

The next step replaces the `Pow` by the subset relation ($\subseteq$).

```
by (resolve_tac [PowI] 1);
  Level 7
  Pow(A Int B) = Pow(A) Int Pow(B)
   1. !!x. [| x : Pow(A); x : Pow(B) |] ==> x <= A Int B
```

We perform the same replacement in the assumptions. This is a good demonstration of the tactic `dresolve_tac`:

```
by (REPEAT (dresolve_tac [PowD] 1));
  Level 8
  Pow(A Int B) = Pow(A) Int Pow(B)
   1. !!x. [| x <= A; x <= B |] ==> x <= A Int B
```

The assumptions are that x is a lower bound of both A and B, but $A \cap B$ is the greatest lower bound:

```
by (resolve_tac [Int_greatest] 1);
  Level 9
  Pow(A Int B) = Pow(A) Int Pow(B)
   1. !!x. [| x <= A; x <= B |] ==> x <= A
   2. !!x. [| x <= A; x <= B |] ==> x <= B
```

To conclude the proof, we clear up the trivial subgoals:

```
by (REPEAT (assume_tac 1));
  Level 10
  Pow(A Int B) = Pow(A) Int Pow(B)
  No subgoals!
```

We could have performed this proof in one step by applying `Blast_tac`. Let us go back to the start:

```
choplev 0;
  Level 0
  Pow(A Int B) = Pow(A) Int Pow(B)
   1. Pow(A Int B) = Pow(A) Int Pow(B)
by (Blast_tac 1);
  Depth = 0
  Depth = 1
  Depth = 2
  Depth = 3
  Level 1
  Pow(A Int B) = Pow(A) Int Pow(B)
  No subgoals!
```

Past researchers regarded this as a difficult proof, as indeed it is if all the symbols are replaced by their definitions.

3.10 Monotonicity of the union operator

For another example, we prove that general union is monotonic: $C \subseteq D$ implies $\bigcup(C) \subseteq \bigcup(D)$. To begin, we tackle the inclusion using `subsetI`:

```
val [prem] = goal thy "C<=D ==> Union(C) <= Union(D)";
  Level 0
  Union(C) <= Union(D)
   1. Union(C) <= Union(D)
  val prem = "C <= D  [C <= D]" : thm
```

```
by (resolve_tac [subsetI] 1);
  Level 1
  Union(C) <= Union(D)
   1. !!x. x : Union(C) ==> x : Union(D)
```

Big union is like an existential quantifier — the occurrence in the assumptions must be eliminated early, since it creates parameters.

```
by (eresolve_tac [UnionE] 1);
  Level 2
  Union(C) <= Union(D)
   1. !!x B. [| x : B; B : C |] ==> x : Union(D)
```

Now we may apply `UnionI`, which creates an unknown involving the parameters. To show $x \in \bigcup(D)$ it suffices to show that x belongs to some element, say $?B2(x, B)$, of D.

```
by (resolve_tac [UnionI] 1);
  Level 3
  Union(C) <= Union(D)
   1. !!x B. [| x : B; B : C |] ==> ?B2(x,B) : D
   2. !!x B. [| x : B; B : C |] ==> x : ?B2(x,B)
```

Combining `subsetD` with the premise $C \subseteq D$ yields $?a \in C \Longrightarrow ?a \in D$, which reduces subgoal 1:

```
by (resolve_tac [prem RS subsetD] 1);
  Level 4
  Union(C) <= Union(D)
   1. !!x B. [| x : B; B : C |] ==> ?B2(x,B) : C
   2. !!x B. [| x : B; B : C |] ==> x : ?B2(x,B)
```

The rest is routine. Note how $?B2(x, B)$ is instantiated.

```
by (assume_tac 1);
  Level 5
  Union(C) <= Union(D)
   1. !!x B. [| x : B; B : C |] ==> x : B
by (assume_tac 1);
  Level 6
  Union(C) <= Union(D)
  No subgoals!
```

Again, `Blast_tac` can prove the theorem in one step, provided we somehow supply it with `prem`. We can add `prem RS subsetD` to the claset as an introduction rule:

```
by (blast_tac (!claset addIs [prem RS subsetD]) 1);
  Depth = 0
  Depth = 1
  Depth = 2
  Level 1
  Union(C) <= Union(D)
  No subgoals!
```

As an alternative, we could add premise to the assumptions, either using `cut_facts_tac` or by stating the original goal using `!!`:

```
goal thy "!!C D. C<=D ==> Union(C) <= Union(D)";
  Level 0
  Union(C) <= Union(D)
   1. !!C D. C <= D ==> Union(C) <= Union(D)
by (Blast_tac 1);
```

The file `ZF/equalities.ML` has many similar proofs. Reasoning about general intersection can be difficult because of its anomalous behaviour on the empty set. However, `Blast_tac` copes well with these. Here is a typical example, borrowed from Devlin [10, page 12]:

```
a:C ==> (INT x:C. A(x) Int B(x)) = (INT x:C.A(x)) Int (INT x:C.B(x))
```

In traditional notation this is

$$a \in C \implies \bigcap_{x\in C} \Bigl(A(x) \cap B(x)\Bigr) = \Bigl(\bigcap_{x\in C} A(x)\Bigr) \cap \Bigl(\bigcap_{x\in C} B(x)\Bigr)$$

3.11 Low-level reasoning about functions

The derived rules `lamI`, `lamE`, `lam_type`, `beta` and `eta` support reasoning about functions in a λ-calculus style. This is generally easier than regarding functions as sets of ordered pairs. But sometimes we must look at the underlying representation, as in the following proof of `fun_disjoint_apply1`. This states that if f and g are functions with disjoint domains A and C, and if $a \in A$, then $(f \cup g)`a = f`a$:

```
val prems = goal thy
    "[| a:A;  f: A->B;  g: C->D;  A Int C = 0 |] ==>  \
\    (f Un g)`a = f`a";
  Level 0
  (f Un g) ` a = f ` a
   1. (f Un g) ` a = f ` a
```

Isabelle has produced the output above; the ML top-level now echoes the binding of `prems`.

```
val prems = ["a : A  [a : A]",
             "f : A -> B  [f : A -> B]",
             "g : C -> D  [g : C -> D]",
             "A Int C = 0  [A Int C = 0]"] : thm list
```

Using `apply_equality`, we reduce the equality to reasoning about ordered pairs. The second subgoal is to verify that $f \cup g$ is a function.

```
by (resolve_tac [apply_equality] 1);
  Level 1
  (f Un g) ` a = f ` a
   1. <a,f ` a> : f Un g
   2. f Un g : (PROD x:?A. ?B(x))
```

We must show that the pair belongs to f or g; by `UnI1` we choose f:

```
by (resolve_tac [UnI1] 1);
  Level 2
  (f Un g) ` a = f ` a
   1. <a,f ` a> : f
   2. f Un g : (PROD x:?A. ?B(x))
```

To show $\langle a, f`a\rangle \in f$ we use `apply_Pair`, which is essentially the converse of `apply_equality`:

```
by (resolve_tac [apply_Pair] 1);
  Level 3
  (f Un g) ` a = f ` a
   1. f : (PROD x:?A2. ?B2(x))
   2. a : ?A2
   3. f Un g : (PROD x:?A. ?B(x))
```

Using the premises $f \in A \to B$ and $a \in A$, we solve the two subgoals from `apply_Pair`. Recall that a Π-set is merely a generalized function space, and observe that `?A2` is instantiated to `A`.

```
by (resolve_tac prems 1);
  Level 4
  (f Un g) ` a = f ` a
   1. a : A
   2. f Un g : (PROD x:?A. ?B(x))
by (resolve_tac prems 1);
  Level 5
  (f Un g) ` a = f ` a
   1. f Un g : (PROD x:?A. ?B(x))
```

To construct functions of the form $f \cup g$, we apply `fun_disjoint_Un`:

```
by (resolve_tac [fun_disjoint_Un] 1);
  Level 6
  (f Un g) ` a = f ` a
   1. f : ?A3 -> ?B3
   2. g : ?C3 -> ?D3
   3. ?A3 Int ?C3 = 0
```

The remaining subgoals are instances of the premises. Again, observe how unknowns are instantiated:

```
by (resolve_tac prems 1);
  Level 7
  (f Un g) ` a = f ` a
   1. g : ?C3 -> ?D3
   2. A Int ?C3 = 0
by (resolve_tac prems 1);
  Level 8
  (f Un g) ` a = f ` a
   1. A Int C = 0
by (resolve_tac prems 1);
  Level 9
  (f Un g) ` a = f ` a
  No subgoals!
```

See the files `ZF/func.ML` and `ZF/WF.ML` for more examples of reasoning about functions.

Chapter 4

Higher-Order Logic

The theory HOL implements higher-order logic. It is based on Gordon's HOL system [16], which itself is based on Church's original paper [6]. Andrews's book [2] is a full description of the original Church-style higher-order logic. Experience with the HOL system has demonstrated that higher-order logic is widely applicable in many areas of mathematics and computer science, not just hardware verification, HOL's original *raison d'être*. It is weaker than ZF set theory but for most applications this does not matter. If you prefer ML to Lisp, you will probably prefer HOL to ZF.

The syntax of HOL[1] follows λ-calculus and functional programming. Function application is curried. To apply the function f of type $\tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3$ to the arguments a and b in HOL, you simply write $f\ a\ b$. There is no 'apply' operator as in ZF. Note that $f(a, b)$ means "f applied to the pair (a, b)" in HOL. We write ordered pairs as (a, b), not $\langle a, b\rangle$ as in ZF.

HOL has a distinct feel, compared with ZF and CTT. It identifies object-level types with meta-level types, taking advantage of Isabelle's built-in type checker. It identifies object-level functions with meta-level functions, so it uses Isabelle's operations for abstraction and application.

These identifications allow Isabelle to support HOL particularly nicely, but they also mean that HOL requires more sophistication from the user — in particular, an understanding of Isabelle's type system. Beginners should work with `show_types` (or even `show_sorts`) set to `true`.

4.1 Syntax

Figure 4.1 lists the constants (including infixes and binders), while Fig. 4.2 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

[1]Earlier versions of Isabelle's HOL used a different syntax. Ancient releases of Isabelle included still another version of HOL, with explicit type inference rules [39]. This version no longer exists, but ZF supports a similar style of reasoning.

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*
`@`	`Eps`	$(\alpha \Rightarrow bool) \Rightarrow \alpha$	Hilbert description (ε)
`!` or `ALL`	`All`	$(\alpha \Rightarrow bool) \Rightarrow bool$	universal quantifier ($\forall$)
`?` or `EX`	`Ex`	$(\alpha \Rightarrow bool) \Rightarrow bool$	existential quantifier ($\exists$)
`?!` or `EX!`	`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 ($\to$)

Infixes

Figure 4.1: Syntax of `HOL`

```
   term  =  expression of class term
         |  @ 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
         |  !   id id* . formula    |  ALL id id* . formula
         |  ?   id id* . formula    |  EX  id id* . formula
         |  ?!  id id* . formula    |  EX! id id* . formula
```

Figure 4.2: Full grammar for HOL

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.

4.1.1 Types and classes

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` offers various methods for introducing new types. See §4.5 and §4.6.

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 partially ordered types (w.r.t. $\leq$).

Three other syntactic type classes — `plus`, `minus` and `times` — permit overloading of the operators `+`, `-` and `*`. In particular, `-` is instantiated for set difference and subtraction on natural numbers.

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.

4.1.2 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 `@`$x . P[x]$.

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 $P\ x\ y$.

Quantifiers have two notations. As in Gordon's HOL system, `HOL` uses `!` and `?` to stand for $\forall$ and $\exists$. The existential quantifier must be followed by a space; thus `?x` is an unknown, while `? x.f x=y` is a quantification. Isabelle's usual notation for quantifiers, `ALL` and `EX`, is also available. Both notations are accepted for input. The ML reference `HOL_quantifiers` governs the output notation. If set to `true`, then `!` and `?` are displayed; this is the default. If set to `false`, then `ALL` and `EX` 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. 4.4). 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

```
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)
selectI        P(x::'a) ==> P(@x.P x)
True_or_False  (P=True) | (P=False)
```

Figure 4.3: The `HOL` rules

least element satisfying P.[2]

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 $P \land (\forall x . Q)$.

4.1.3 The `let` and `case` constructions

Local abbreviations can be introduced by a `let` construct whose syntax appears in Fig. 4.2. 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

`case` e `of` c_1 `=>` e_1 `|` $\ldots$ `|` c_n `=>` 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 §4.6).

! 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 $f\ x =$ `if`...`then`...`else`... you need to write $f\ x =$ (`if`...`then`...`else`...).

4.2 Rules of inference

Figure 4.3 shows the primitive inference rules of `HOL`, with their ML names. Some of the rules deserve additional comments:

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

```
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 4.4: The HOL definitions

`ext` expresses extensionality of functions.

`iff` asserts that logically equivalent formulae are equal.

`selectI` gives the defining property of the Hilbert ε-operator. It is a form of the Axiom of Choice. The derived rule `select_equality` (see below) is often easier to use.

`True_or_False` makes the logic classical.[3]

HOL follows standard practice in higher-order logic: only a few connectives are taken as primitive, with the remainder defined obscurely (Fig. 4.4). Gordon's HOL system expresses the corresponding definitions [16, 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.

! 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 (see §4.4).

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 4.5 and 4.6, 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.

[3]In fact, the ε-operator already makes the logic classical, as shown by Diaconescu; see Paulson [39] for details.

```
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 4.5: 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

select_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
expand_if       P(if Q then x else y) = ((Q --> P x) & (~Q --> P y))
```

Conditionals

Figure 4.6: More derived rules

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.

4.3 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.

4.3.1 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 4.7 lists the constants, infixes, and syntax translations. Figure 4.8 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 $\neg(a \in b)$.

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
`Compl`	$\alpha\,set \Rightarrow \alpha\,set$	complement
`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 over a type
`UN`	`UNION1`	$(\alpha \Rightarrow \beta\,set) \Rightarrow \beta\,set$	10	union over a type

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 4.7: Syntax of the theory `Set`

external	*internal*	*description*
`a ~: b`	`~(a : b)`	non-membership
`{`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
`!` $x{:}A.P[x]$ `or ALL` $x{:}A.P[x]$	`Ball` A $\lambda x\,.\,P[x]$	bounded $\forall$
`?` $x{:}A.P[x]$ `or EX` $x{:}A.P[x]$	`Bex` A $\lambda x\,.\,P[x]$	bounded $\exists$

Translations

---

term	=	other terms…
	|	`{}`
	|	`{` *term* `(,`*term*`)`* `}`
	|	`{` *id* `.` *formula* `}`
	|	*term* ` `` ` *term*
	|	*term* `Int` *term*
	|	*term* `Un` *term*
	|	`INT` *id*`:`*term* `.` *term*
	|	`UN` *id*`:`*term* `.` *term*
	|	`INT` *id id** `.` *term*
	|	`UN` *id id** `.` *term*
formula	=	other formulae…
	|	*term* `:` *term*
	|	*term* `~:` *term*
	|	*term* `<=` *term*
	|	`!` *id*`:`*term* `.` *formula* | `ALL` *id*`:`*term* `.` *formula*
	|	`?` *id*`:`*term* `.` *formula* | `EX` *id*`:`*term* `.` *formula*

Full Grammar

Figure 4.8: Syntax of the theory `Set` (continued)

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 `Collect`$(\lambda x . P[x])$. It defines sets by absolute comprehension, which is impossible in `ZF`; the type of x implicitly restricts the comprehension.

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

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

The constants `Ball` and `Bex` are defined accordingly. Instead of `Ball` A P and `Bex` A P we may write `!` $x\!:\!A . P[x]$ and `?` $x\!:\!A . P[x]$. Isabelle's usual quantifier symbols, `ALL` and `EX`, are also accepted for input. As with the primitive quantifiers, the ML reference `HOL_quantifiers` specifies which notation to use for output.

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.

4.3.2 Axioms and rules of set theory

Figure 4.9 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.

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 4.10 and 4.11 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 4.12 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`.

```
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         Compl 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 4.9: Rules of the theory `Set`

Figure 4.13 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.

4.3.3 Properties of functions

Figure 4.14 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 (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`.

4.4 Generic packages

`HOL` instantiates most of Isabelle's generic packages, making available the simplifier and the classical reasoner.

```
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 4.10: 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 : Compl A
ComplD   [| c : Compl 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 4.11: 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 4.12: Derived rules involving subsets

```
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 (Compl A) = {x.False}
Compl_partition    A Un  (Compl A) = {x.True}
double_complement  Compl(Compl A) = A
Compl_Un           Compl(A Un B)  = (Compl A) Int (Compl B)
Compl_Int          Compl(A Int B) = (Compl A) Un (Compl 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)
Un_Union_image     (UN x:C.(A x) Un (B x)) = Union(A``C) Un Union(B``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)
Int_Inter_image    (INT x:C.(A x) Int (B x)) = Inter(A``C) Int Inter(B``C)
```

Figure 4.13: Set equalities

name	*meta-type*	*description*
`inj surj`	$(\alpha \Rightarrow \beta) \Rightarrow bool$	injective/surjective
`inj_onto`	$[\alpha \Rightarrow \beta, \alpha\, set] \Rightarrow bool$	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_onto_def     inj_onto f A == !x:A. !y:A. f x=f y --> x=y
inv_def          inv f        == (%y. @x. f(x)=y)
```

Figure 4.14: Theory `Fun`

4.4.1 Simplification and substitution

The simplifier is available in `HOL`. 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 $a = b \wedge P(a)$ to $a = b \wedge P(b)$ 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]`.

If the simplifier cannot use a certain rewrite rule — either because of non-termination 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.

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 `expand_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` (analogous to `addsimps`) of type `simpset * thm list -> simpset` that adds rules such as $(*)$ to a simpset, as in

```
by(simp_tac (!simpset addsplits [expand_if]) 1);
```

The effect is that after each round of simplification, one occurrence of `if` is split acording to `expand_if`, until all occurences of `if` have been eliminated.

In general, `addsplits` accepts rules of the form

$$?P(c\ ?x_1\ \ldots\ ?x_n)\ =\ \mathit{rhs}$$

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 §4.5.3 and §4.6.1).

4.4.2 Classical reasoning

`HOL` derives classical introduction rules for $\lor$ and $\exists$, as well as classical elimination rules for $\to$ and $\leftrightarrow$, and the swap rule; recall Fig. 4.6 above.

The classical reasoner is installed. Tactics such as `Blast_tac` and `Best_tac` use the default claset (`!claset`), which works for most purposes. Named clasets include `prop_cs`, which includes the propositional rules, and `HOL_cs`, which also includes quantifier rules. See the file `HOL/cladata.ML` for lists of the classical rules, and the *Reference Manual* for more discussion of classical proof methods.

4.5 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 §4.6.

4.5.1 Product and sum types

Theory `Prod` (Fig. 4.15) defines the product type $\alpha \times \beta$, with the ordered pair syntax (a, b). General tuples are simulated by pairs nested to the right:

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)}

Pair_eq      ((a,b) = (a',b')) = (a=a' & b=b')
Pair_inject  [| (a, b) = (a',b');  [| a=a';  b=b' |] ==> R |] ==> R
PairE        [| !!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        split c (a,b) = c a b
expand_split R(split 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 4.15: Type $\alpha\times\beta$

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: `%(`x`,`y`,`z`).`t $\leadsto$ `%(`x`,(`y`,`z`)).`t $\leadsto$ `split(%`x`.%(`y`,`z`).`t`)` $\leadsto$ `split(%`x`.split(%`y z`.`t`))`. The reverse translation is performed upon printing.

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

In addition to explicit 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: `!` *pattern*`:`A`.` P

Choice: `@` *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. 4.16) 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.

symbol	*meta-type*	*description*
`Inl`	$\alpha \Rightarrow \alpha + \beta$	first injection
`Inr`	$\beta \Rightarrow \alpha + \beta$	second injection
`sum_case`	$[\alpha \Rightarrow \gamma, \beta \Rightarrow \gamma, \alpha + \beta] \Rightarrow \gamma$	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

sum_case_Inl   sum_case f g (Inl x) = f x
sum_case_Inr   sum_case f g (Inr x) = g x

surjective_sum sum_case (%x. f(Inl x)) (%y. f(Inr y)) s = f s
expand_sum_case R(sum_case f g s) = ((! x. s = Inl(x) --> R(f(x))) &
                                     (! y. s = Inr(y) --> R(g(y))))
```

Figure 4.16: Type $\alpha + \beta$

symbol	*meta-type*	*priority*	*description*
`0`	*nat*		zero
`Suc`	$nat \Rightarrow nat$		successor function
`*`	$[nat, nat] \Rightarrow nat$	Left 70	multiplication
`div`	$[nat, nat] \Rightarrow nat$	Left 70	division
`mod`	$[nat, nat] \Rightarrow nat$	Left 70	modulus
`+`	$[nat, nat] \Rightarrow nat$	Left 65	addition
`-`	$[nat, nat] \Rightarrow nat$	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 4.17: 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 4.18: Recursion equations for the arithmetic operators

4.5.2 The type of natural numbers, *nat*

The theory `NatDef` 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. For details see the file `NatDef.thy`.

Type *nat* is an instance of class `ord`, which makes the overloaded functions of this class (esp. `<` and `<=`, but also `min`, `max` and `LEAST`) available on *nat*. Theory `Nat` builds on `NatDef` and shows that `<=` is a partial order, so *nat* is also an instance of class `order`.

Theory `Arith` 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. 4.17 and 4.18. 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 §4.7. A simple example is addition. Here, `op +` is the name of the infix operator `+`, following the standard convention.

```
primrec "op +" nat
  "    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 `nat_rec`, the details of which can be found in theory `NatDef`.

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
```

Reasoning about arithmetic inequalities can be tedious. A minimal amount of automation is provided by the tactic `trans_tac` of type `int -> tactic` that deals with simple inequalities. Note that it only knows about `0`, `Suc`, `<` and `<=`. The following goals are all solved by `trans_tac 1`:

```
1. ... ==> m <= Suc(Suc m)
1. [| ... i <= j ... Suc j <= k ... |] ==> i < k
1. [| ... Suc m <= n ... ~ m < n ... |] ==> ...
```

For a complete description of the limitations of the tactic and how to avoid some of them, see the comments at the start of the file `Provers/nat_transitive.ML`.

If `trans_tac` fails you, try to find relevant arithmetic results in the library. The theory `NatDef` contains theorems about `<` and `<=`, the theory `Arith` contains theorems about `+`, `-` and `*`, and theory `Divides` contains theorems about `div` and `mod`. Use the `find`-functions to locate them (see the *Reference Manual*).

4.5.3 The type constructor for lists, *list*

Figure 4.19 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 `exhaust_tac` also become available for lists. A `case` construct of the form

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

is defined by translation. For details see §4.6. There is also a case splitting rule `split_list_case`

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

which can be fed to `addsplits` just like `expand_if` (see §4.4.1).

`List` provides a basic library of list processing functions defined by primitive recursion (see §4.7.1). The recursion equations are shown in Fig. 4.20.

4.5.4 Introducing new types

The `HOL`-methodology dictates that all extensions to a theory should be **definitional**. The type definition mechanism that meets this criterion is `typedef`.

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`	$[\alpha, \alpha\, list] \Rightarrow bool$	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
`nth`	$nat \Rightarrow \alpha\, list \Rightarrow \alpha$		indexing
`take`, `drop`	$nat \Rightarrow \alpha\, list \Rightarrow \alpha\, list$		take or drop a prefix
`takeWhile`, `dropWhile`	$(\alpha \Rightarrow bool) \Rightarrow \alpha\, list \Rightarrow \alpha\, list$		take or drop a prefix

Constants and infixes

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

Translations

Figure 4.19: 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

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)

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

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

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

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

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

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

nth 0 xs = hd xs
nth (Suc n) xs = nth n (tl 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 4.20: Recursions equations for list processing functions

Note that *type synonyms*, which are inherited from `Pure` and described elsewhere, are just syntactic abbreviations that have no logical meaning.

! Types in `HOL` must be non-empty; otherwise the quantifier rules would be unsound, because $\exists x \,.\, x = x$ is a theorem [39, §7].

A **type definition** identifies the new type with a subset of an existing type. More precisely, the new type is defined by exhibiting an existing type τ, a set $A :: \tau\, set$, and a theorem of the form $x : A$. Thus A is a non-empty subset of τ, and the new type denotes this subset. New functions are defined that establish an isomorphism between the new type and the subset. If type τ involves type variables $\alpha_1, \ldots, \alpha_n$, then the type definition creates a type constructor $(\alpha_1, \ldots, \alpha_n)ty$ rather than a particular type.

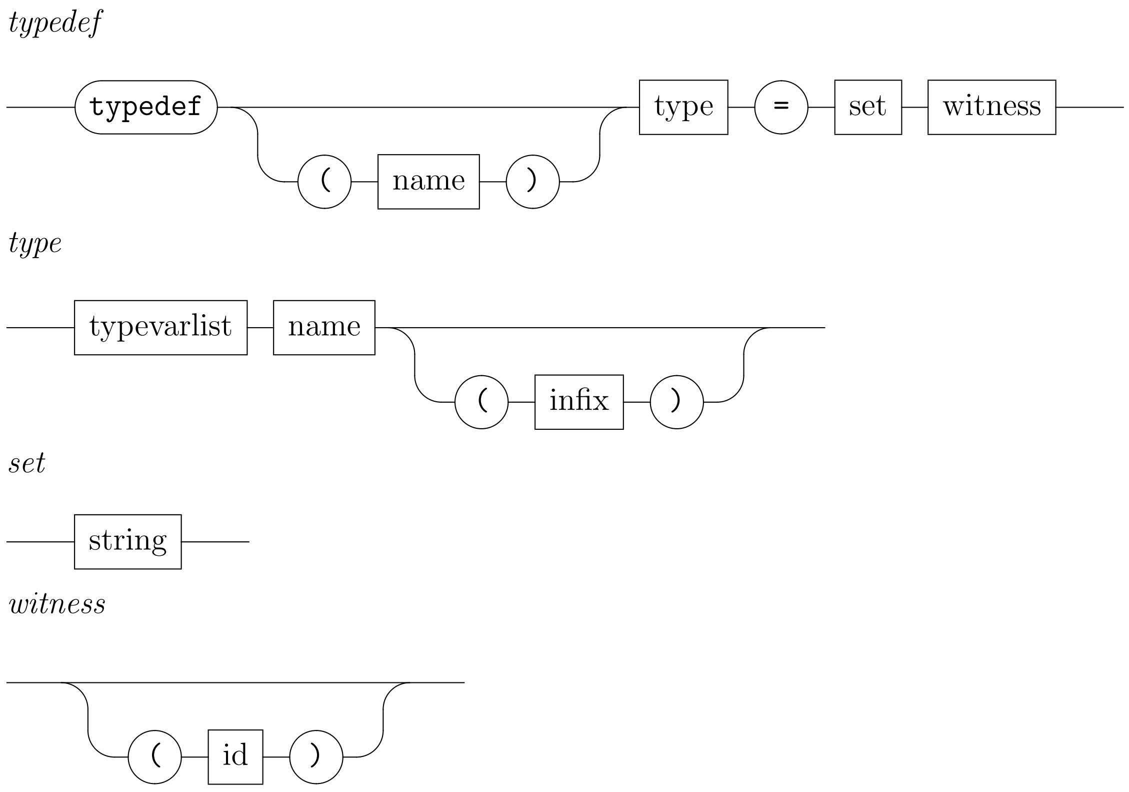

Figure 4.21: Syntax of type definitions

The syntax for type definitions is shown in Fig. 4.21. For the definition of ‘typevarlist’ and ‘infix’ see the appendix of the *Reference Manual*. The remaining nonterminals have the following meaning:

type: the new type constructor $(\alpha_1, \ldots, \alpha_n)ty$ with optional infix annotation.

name: an alphanumeric name T for the type constructor ty, in case ty is a symbolic name. Defaults to ty.

set: the representing subset A.

witness: name of a theorem of the form $a : A$ proving non-emptiness. It can be omitted in case Isabelle manages to prove non-emptiness automatically.

If all context conditions are met (no duplicate type variables in 'typevarlist', no extra type variables in 'set', and no free term variables in 'set'), the following components are added to the theory:

- a type $ty :: (term, \dots, term)term$
- constants

$$\begin{array}{rcl} T & :: & \tau\ set \\ Rep\_T & :: & (\alpha_1, \dots, \alpha_n)ty \Rightarrow \tau \\ Abs\_T & :: & \tau \Rightarrow (\alpha_1, \dots, \alpha_n)ty \end{array}$$

- a definition and three axioms

T`_def`	$T \equiv A$
`Rep_`T	$Rep\_T\, x \in T$
`Rep_`T`_inverse`	$Abs\_T\,(Rep\_T\, x) = x$
`Abs_`T`_inverse`	$y \in T \Longrightarrow Rep\_T\,(Abs\_T\, y) = y$

  stating that $(\alpha_1, \dots, \alpha_n)ty$ is isomorphic to A by Rep_T and its inverse Abs_T.

Below are two simple examples of `HOL` type definitions. Non-emptiness is proved automatically here.

```
typedef unit = "{True}"

typedef (prod)
  ('a, 'b) "*"    (infixr 20)
      = "{f . EX (a::'a) (b::'b). f = (%x y. x = a & y = b)}"
```

Type definitions permit the introduction of abstract data types in a safe way, namely by providing models based on already existing types. Given some abstract axiomatic description P of a type, this involves two steps:

1. Find an appropriate type τ and subset A which has the desired properties P, and make a type definition based on this representation.
2. Prove that P holds for ty by lifting P from the representation.

You can now forget about the representation and work solely in terms of the abstract properties P.

! If you introduce a new type (constructor) *ty* axiomatically, i.e. by declaring the type and its operations and by stating the desired axioms, you should make sure the type has a non-empty model. You must also have a clause

```
arities ty :: (term,...,term) term
```

in your theory file to tell Isabelle that *ty* is in class `term`, the class of all `HOL` types.

4.6 Datatype declarations

Inductive datatypes, similar to those of ML, frequently appear in non-trivial applications of `HOL`. In principle, such types could be defined by hand via `typedef` (see §4.5.4), but this would be far too tedious. The `datatype` definition package of `HOL` automates such chores. It generates freeness theorems and induction rules from a very simple description of the new type provided by the user.

4.6.1 Basics

The general `HOL` `datatype` definition is of the following form:

$$\texttt{datatype}\ (\alpha_1,\ldots,\alpha_n)\,t\ =\ C_1\ \tau_{11}\ \ldots\ \tau_{1k_1}\ \mid\ \ldots\ \mid\ C_m\ \tau_{m1}\ \ldots\ \tau_{mk_m}$$

where α_i are type variables, C_i are distinct constructor names and τ_{ij} are types. The latter may be one of the following:

- type variables $\alpha_1,\ldots,\alpha_n$,
- types $(\beta_1,\ldots,\beta_l)\,t'$ where t' is a previously declared type constructor or type synonym and $\{\beta_1,\ldots,\beta_l\} \subseteq \{\alpha_1,\ldots,\alpha_n\}$,
- the newly defined type $(\alpha_1,\ldots,\alpha_n)\,t$.

Recursive occurences of $(\alpha_1,\ldots,\alpha_n)\,t$ are quite restricted. To ensure that the new type is non-empty, at least one constructor must consist of only non-recursive type components. If you would like one of the τ_{ij} to be a complex type expression τ you need to declare a new type synonym $syn = \tau$ first and use syn in place of τ. Of course this does not work if τ mentions the recursive type itself, thus ruling out problematic cases like `datatype` $t\ =\ C\,(t \Rightarrow t)$, but also unproblematic ones like `datatype` $t\ =\ C\,(t\ list)$.

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

$$C_j :: [\tau_{j1},\ldots,\tau_{jk_j}] \Rightarrow (\alpha_1,\ldots,\alpha_n)t$$

These functions have certain *freeness* properties — they are distinct:

$$C_i\ x_1\ \ldots\ x_{k_i} \neq C_j\ y_1\ \ldots\ y_{k_j} \qquad \text{for all } i \neq j.$$

and they are injective:

$$(C_j\ x_1\ \ldots\ x_{k_j} = C_j\ y_1\ \ldots\ y_{k_j}) = (x_1 = y_1 \land \ldots \land x_{k_j} = y_{k_j})$$

Because the number of inequalities is quadratic in the number of constructors, a different representation is used if there are 7 or more of them. In that case every constructor term is mapped to a natural number:

$$t\_ord\,(C_i\, x_1\ \ldots\ x_{k_i}) = i - 1$$

Then distinctness of constructor terms is expressed by:

$$t\_ord\, x \neq t\_ord\, y \Longrightarrow x \neq y.$$

Generally, the following structural induction rule is provided:

$$\frac{\begin{array}{lcl} \bigwedge x_1 \ldots x_{k_1} .\, [\![P\ x_{r_{11}}; \ldots ; P\ x_{r_{1l_1}}]\!] & \Longrightarrow & P\,(C_1\ x_1\ \ldots\ x_{k_1}) \\ & \vdots & \\ \bigwedge x_1 \ldots x_{k_m} .\, [\![P\ x_{r_{m1}}; \ldots ; P\ x_{r_{ml_m}}]\!] & \Longrightarrow & P\,(C_m\ x_1\ \ldots\ x_{k_m}) \end{array}}{P\, x}$$

where $\{r_{j1}, \ldots, r_{jl_j}\} = \{i \in \{1, \ldots k_j\} \ \mid\ \tau_{ji} = (\alpha_1, \ldots, \alpha_n)t\} =: Rec_j$, i.e. the property P can be assumed for all arguments of the recursive type.

For convenience, the following additional constructions are predefined for each datatype.

The `case` construct

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

$$\begin{array}{lllcl} \texttt{case}\ e\ \texttt{of} & & C_1\ x_{11}\ \ldots\ x_{1k_1} & \Rightarrow & e_1 \\ & \vdots & & & \\ & \mid & C_m\ x_{m1}\ \ldots\ x_{mk_m} & \Rightarrow & e_m \end{array}$$

where the x_{ij} are either identifiers or nested tuple patterns as in §4.5.1.

! In contrast to ML, *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 `split_`t`_casesplit_`t`_case` is provided:

$$\begin{array}{lcl} P(t\texttt{\_case}\ f_1\ \ldots\ f_m\ e) & = & ((\forall x_1 \ldots x_{k_1} .\ e = C_1\ x_1 \ldots x_{k_1} \to P(f_1\ \ x_1 \ldots x_{k_1})) \\ & & \ \land\ \ldots\ \land \\ & & \ (\forall x_1 \ldots x_{k_m} .\ e = C_m\ x_1 \ldots x_{k_m} \to P(f_m\ \ x_1 \ldots x_{k_m}))) \end{array}$$

where t`_case` is the internal name of the `case`-construct. This theorem can be added to a simpset via `addsplits` (see §4.4.1).

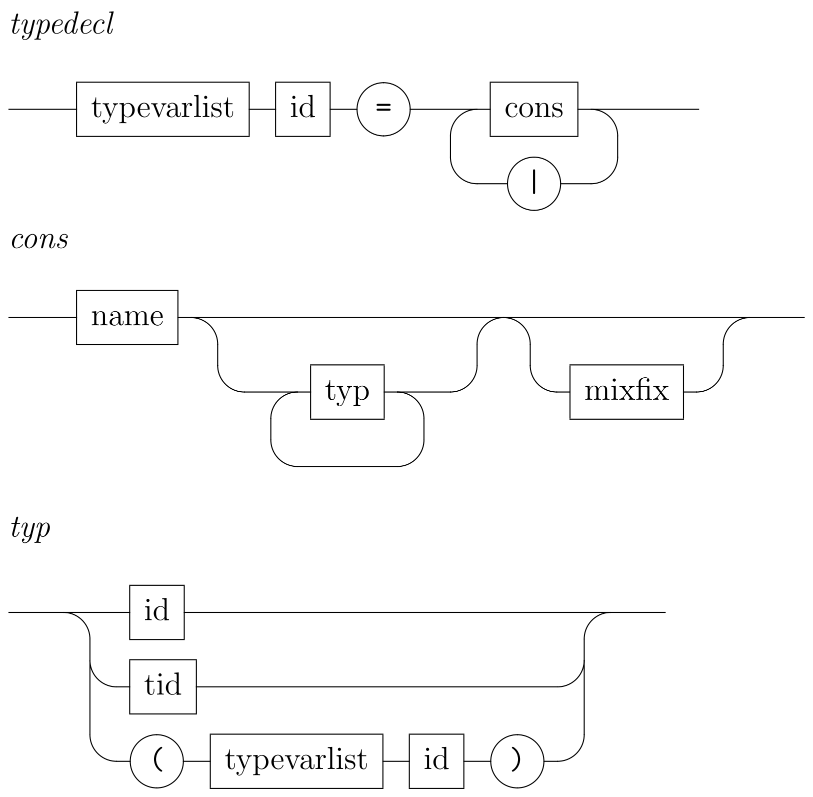

Figure 4.22: Syntax of datatype declarations

The function `size`

Theory `Arith` declares an overloaded function `size` of type $\alpha \Rightarrow nat$. Each datatype defines a particular instance of `size` according to the following scheme:

$$size(C_j\ x_{j1}\ \dots\ x_{jk_1}) = \begin{cases} 0 & \text{if } Rec_j = \emptyset \\ size(x_{r_{j1}}) + \cdots + size(x_{r_{jl_j}}) + 1 & \text{if } Rec_j = \{r_{j1}, \dots, r_{jl_j}\} \end{cases}$$

where Rec_j is defined above. Viewing datatypes as generalized trees, the size of a leaf is 0 and the size of a node is the sum of the sizes of its subtrees $+1$.

4.6.2 Defining datatypes

A datatype is defined in a theory definition file using the keyword `datatype`. The definition following this must conform to the syntax of *typedecl* specified in Fig. 4.22 and must obey the rules in the previous section. As a result the theory is extended with the new type, the constructors, and the theorems listed in the previous section.

! Every theory containing a datatype declaration must be based, directly or indirectly, on the theory `Arith`, if necessary by including it explicitly as a parent.

Most of the theorems about the datatype become part of the default simpset and you never need to see them again because the simplifier applies them automatically. Only induction is invoked by hand:

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

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

`exhaust_tac "`*u*`"` *i* performs an exhaustive case analysis for the term u whose type must be a datatype or type *nat*. If the datatype has n constructors C_1, … C_n, subgoal i is replaced by n new subgoals which contain the additional assumption $u = C_j\ x_1\ \ldots\ x_{k_j}$ for $j = 1, \ldots, n$.

! Induction is only allowed on a free variable that should not occur among the premises of the subgoal. Exhaustion works for arbitrary terms.

For the technically minded, we give a more detailed description. Reading 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 cases : thm list
val simps : thm list
val induct_tac : string -> int -> tactic
```

`distinct`, `inject` and `induct` 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`.

4.6.3 Examples

The datatype *α mylist*

We want to define the type *α mylist*.[4] To do this we have to build a new theory that contains the type definition. We start from the basic `HOL` theory.

```
MyList = HOL +
  datatype 'a mylist = Nil | Cons 'a ('a mylist)
end
```

[4]This is just an example, there is already a list type in `HOL`, of course.

After loading the theory (with `use_thy "MyList"`), we can prove *Cons x xs* $\neq$ *xs*. To ease the induction applied below, we state the goal with x quantified at the object-level. This will be stripped later using `qed_spec_mp`.

```
goal MyList.thy "!x. Cons x xs ~= xs";
  Level 0
  ! x. Cons x xs ~= xs
   1. ! x. Cons x xs ~= xs
```

This can be proved by the structural induction tactic:

```
by (induct_tac "xs" 1);
  Level 1
  ! x. Cons x xs ~= xs
   1. ! x. Cons x Nil ~= Nil
   2. !!a mylist.
         ! x. Cons x mylist ~= mylist ==>
         ! x. Cons x (Cons a mylist) ~= Cons a mylist
```

The first subgoal can be proved using the simplifier. Isabelle has already added the freeness properties of lists to the default simplification set.

```
by (Simp_tac 1);
  Level 2
  ! x. Cons x xs ~= xs
   1. !!a mylist.
         ! x. Cons x mylist ~= mylist ==>
         ! x. Cons x (Cons a mylist) ~= Cons a mylist
```

Similarly, we prove the remaining goal.

```
by (Asm_simp_tac 1);
  Level 3
  ! x. Cons x xs ~= xs
  No subgoals!
qed_spec_mp "not_Cons_self";
  val not_Cons_self = "Cons x xs ~= xs";
```

Because both subgoals could have been proved by `Asm_simp_tac` we could have done that in one step:

```
by (ALLGOALS Asm_simp_tac);
```

The datatype α *mylist* with mixfix syntax

In this example we define the type α *mylist* again but this time we want to write `[]` for `Nil` and we want to use infix notation `#` for `Cons`. To do this we simply add mixfix annotations after the constructor declarations as follows:

```
MyList = HOL +
  datatype 'a mylist =
    Nil ("[]")  |
    Cons 'a ('a mylist)  (infixr "#" 70)
end
```

Now the theorem in the previous example can be written `x#xs ~= xs`. The proof is the same.

A datatype for weekdays

This example shows a datatype that consists of 7 constructors:

```
Days = Arith +
  datatype days = Mon | Tue | Wed | Thu | Fri | Sat | Sun
end
```

Because there are more than 6 constructors, the theory must be based on `Arith`. Inequality is expressed via a function `days_ord`. The theorem `Mon ~= Tue` is not directly contained among the distinctness theorems, but the simplifier can prove it thanks to rewrite rules inherited from theory `Arith`:

```
goal Days.thy "Mon ~= Tue";
by (Simp_tac 1);
```

You need not derive such inequalities explicitly: the simplifier will dispose of them automatically.

4.7 Recursive function definitions

Isabelle/HOL provides two means of declaring recursive functions.

- **Primitive recursion** is available only for datatypes, and it is highly restrictive. Recursive calls are only allowed on the argument's immediate constituents. On the other hand, it is the form of recursion most often wanted, and it is easy to use.

- **Well-founded recursion** 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.[5]

[5]This facility is based on Konrad Slind's TFL package [44]. Thanks are due to Konrad for implementing TFL and assisting with its installation.

A theory file may contain any number of recursive function definitions, which may be intermixed with other declarations. Every recursive function must already have been declared as a constant.

These declarations do not assert new axioms. Instead, they define the function using a recursion operator. Both HOL and ZF derive the theory of well-founded recursion from first principles [35]. Primitive recursion over some datatype relies on the recursion operator provided by the datatype package. With either form of function definition, Isabelle proves the desired recursion equations as theorems.

4.7.1 Primitive recursive functions

Datatypes come with a uniform way of defining functions, **primitive recursion**. In principle, one can define primitive recursive functions by asserting their reduction rules as new axioms. Here is an example:

```
Append = MyList +
consts app :: ['a mylist, 'a mylist] => 'a mylist
rules
   app_Nil   "app [] ys = ys"
   app_Cons  "app (x#xs) ys = x#app xs ys"
end
```

But asserting axioms brings the danger of accidentally asserting an inconsistency, as in `app [] ys = us`.

The `primrec` declaration is a safe means of defining primitive recursive functions on datatypes:

```
Append = MyList +
consts app :: ['a mylist, 'a mylist] => 'a mylist
primrec app MyList.mylist
   "app [] ys = ys"
   "app (x#xs) ys = x#app xs ys"
end
```

Isabelle will now check that the two rules do indeed form a primitive recursive definition, preserving consistency. For example

```
primrec app MyList.mylist
    "app [] ys = us"
```

is rejected with an error message `Extra variables on rhs`.

The general form of a primitive recursive definition is

`primrec` *function type*
     reduction rules

where

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

- *type* is the name of the datatype, either as an *id* or in the long form $T.t$ (T is the name of the theory where the datatype has been declared, t the name of the datatype). The long form is required if the `datatype` and the `primrec` sections are in different theories.

- *reduction rules* specify one or more equations of the form

$$f\,x_1\,\ldots\,x_m\,(C\,y_1\,\ldots\,y_k)\,z_1\,\ldots\,z_n = r$$

  such that C is a constructor of the datatype, r contains only the free variables on the left-hand side, and all recursive calls in r are of the form $f\,\ldots\,y_i\,\ldots$ for some i. There must be exactly one reduction rule for each constructor. The order is immaterial. Also note that all reduction rules are added to the default simpset!

  If you would like to refer to some rule by name, then you must prefix *each* rule with an identifier. These identifiers, like those in the `rules` section of a theory, will be visible at the ML level.

The primitive recursive function can have infix or mixfix syntax:

```
Append = MyList +
consts "@"  :: ['a mylist, 'a mylist] => 'a mylist  (infixr 60)
primrec "op @" MyList.mylist
   "[] @ ys = ys"
   "(x#xs) @ ys = x#(xs @ ys)"
end
```

The reduction rules for `@` become part of the default simpset, which leads to short proofs:

```
goal Append.thy "(xs @ ys) @ zs = xs @ (ys @ zs)";
by (induct_tac "xs" 1);
by (ALLGOALS Asm_simp_tac);
```

4.7.2 Well-founded recursive functions

Well-founded recursion can express any function whose termination can be proved by showing that each recursive call makes the argument smaller in a suitable sense. The recursion need not involve datatypes and there are few syntactic restrictions. Nested recursion and pattern-matching are allowed.

Here is a simple example, the Fibonacci function. The first line declares `fib` to be a constant. The well-founded relation is simply $<$ (on the natural numbers).

Pattern-matching is used here: `1` is a macro for `Suc 0`.

```
consts fib  :: "nat => nat"
recdef fib "less_than"
    "fib 0 = 0"
    "fib 1 = 1"
    "fib (Suc(Suc x)) = (fib x + fib (Suc x))"
```

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 $\prec$ should be a relation over τ: it must have type $(\tau \times \tau)set$.

Proving well-foundedness can be tricky, so `HOL` provides a collection of operators for building well-founded relations. The package recognizes 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 $x \prec y$ iff $f(x) < f(y)$. 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 §4.6.1).
- `inv_image` f R is a generalization of `measure`. It specifies a relation such that $x \prec y$ iff $f(x)$ is less than $f(y)$ according to R, which must itself be a well-founded relation.
- R_1`**`R_2 is the lexicographic product of two relations. It is a relation on pairs and satisfies $(x_1, x_2) \prec (y_1, y_2)$ iff 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 primitive 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 patterns must be exhaustive and non-overlapping.

  Unlike with `primrec`, the reduction rules are not added to the default simpset, and individual rules may not be labelled with identifiers. However, the identifier *f*`.rules` is visible at the ML level as a list of theorems.

With the definition of `gcd` shown above, Isabelle is unable to prove one termination condition. It remains as a precondition of the recursion theorems.

```
gcd.rules;
  ["! 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 `Primes` (on the examples directory `HOL/ex`) 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*`.rules` and its effect is to set up a proof of the termination conditions:

```
Tfl.tgoalw thy [] gcd.rules;
  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.rules`. Theory `Unify` on directory `HOL/Subst` 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 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))"
```

A `recdef` definition also returns an induction rule specialized 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 be additional premises of this rule.

4.8 Inductive and coinductive definitions

An **inductive definition** specifies the least set R closed under given rules. (Applying a rule to elements of R yields a result within R.) For example, a structural operational semantics is an inductive definition of an evaluation relation. Dually, a **coinductive definition** specifies the greatest set R consistent with given rules. (Every element of R can be seen as arising by applying a rule to elements of R.) An important example is using bisimulation relations to formalize equivalence of processes and infinite data structures.

A theory file may contain any number of inductive and coinductive definitions. They may be intermixed with other declarations; in particular, the (co)inductive sets **must** be declared separately as constants, and may have mixfix syntax or be subject to syntax translations.

Each (co)inductive definition adds definitions to the theory and also proves some theorems. Each definition creates an ML structure, which is a substructure of the main theory structure.

This package is derived from the `ZF` one, described in a separate paper,[6] which you should refer to in case of difficulties. The package is simpler than `ZF`'s thanks to `HOL`'s automatic type-checking. The type of the (co)inductive determines the

[6]It appeared in CADE [34]; a longer version is distributed with Isabelle.

domain of the fixedpoint definition, and the package does not use inference rules for type-checking.

4.8.1 The result structure

Many of the result structure's components have been discussed in the paper; others are self-explanatory.

`thy` is the new theory containing the recursive sets.

`defs` is the list of definitions of the recursive sets.

`mono` is a monotonicity theorem for the fixedpoint operator.

`unfold` is a fixedpoint equation for the recursive set (the union of the recursive sets, in the case of mutual recursion).

`intrs` is the list of introduction rules, now proved as theorems, for the recursive sets. The rules are also available individually, using the names given them in the theory file.

`elim` is the elimination rule.

`mk_cases` is a function to create simplified instances of `elim`, using freeness reasoning on some underlying datatype.

For an inductive definition, the result structure contains two induction rules, `induct` and `mutual_induct`. (To save storage, the latter rule is just `True` unless more than one set is being defined.) For a coinductive definition, it contains the rule `coinduct`.

Figure 4.23 summarizes the two result signatures, specifying the types of all these components.

4.8.2 The syntax of a (co)inductive definition

An inductive definition has the form

```
inductive    inductive sets
  intrs      introduction rules
  monos      monotonicity theorems
  con_defs   constructor definitions
```

A coinductive definition is identical, except that it starts with the keyword `coinductive`.

The `monos` and `con_defs` sections are optional. If present, each is specified as a string, which must be a valid ML expression of type `thm list`. It is simply inserted into the generated ML file that is generated from the theory definition;

```
sig
val thy          : theory
val defs         : thm list
val mono         : thm
val unfold       : thm
val intrs        : thm list
val elim         : thm
val mk_cases     : thm list -> string -> thm
(Inductive definitions only)
val induct       : thm
val mutual_induct: thm
(Coinductive definitions only)
val coinduct    : thm
end
```

Figure 4.23: The result of a (co)inductive definition

if it is ill-formed, it will trigger ML error messages. You can then inspect the file on your directory.

- The *inductive sets* are specified by one or more strings.
- The *introduction rules* specify one or more introduction rules in the form *ident string*, where the identifier gives the name of the rule in the result structure.
- The *monotonicity theorems* are required for each operator applied to a recursive set in the introduction rules. There **must** be a theorem of the form $A \subseteq B \Longrightarrow M(A) \subseteq M(B)$, for each premise $t \in M(R_i)$ in an introduction rule!
- The *constructor definitions* contain definitions of constants appearing in the introduction rules. In most cases it can be omitted.

The package has a few notable restrictions:

- The theory must separately declare the recursive sets as constants.
- The names of the recursive sets must be alphanumeric identifiers.
- Side-conditions must not be conjunctions. However, an introduction rule may contain any number of side-conditions.
- Side-conditions of the form $x = t$, where the variable x does not occur in t, will be substituted through the rule `mutual_induct`.

4.8.3 Example of an inductive definition

Two declarations, included in a theory file, define the finite powerset operator. First we declare the constant Fin. Then we declare it inductively, with two introduction rules:

```
consts Fin :: 'a set => 'a set set
inductive "Fin A"
  intrs
    emptyI  "{} : Fin A"
    insertI "[| a: A;  b: Fin A |] ==> insert a b : Fin A"
```

The resulting theory structure contains a substructure, called Fin. It contains the Fin A introduction rules as the list Fin.intrs, and also individually as Fin.emptyI and Fin.consI. The induction rule is Fin.induct.

For another example, here is a theory file defining the accessible part of a relation. The main thing to note is the use of Pow in the sole introduction rule, and the corresponding mention of the rule Pow_mono in the monos list. The paper [34] discusses a ZF version of this example in more detail.

```
Acc = WF +
consts pred :: "['b, ('a * 'b)set] => 'a set"   (*Set of predecessors*)
       acc  :: "('a * 'a)set => 'a set"         (*Accessible part*)
defs   pred_def  "pred x r == y. (y,x):r"
inductive "acc r"
  intrs
     pred "pred a r: Pow(acc r) ==> a: acc r"
  monos   "[Pow_mono]"
end
```

The HOL distribution contains many other inductive definitions. Simple examples are collected on subdirectory Induct. The theory HOL/Induct/LList.thy contains coinductive definitions. Larger examples may be found on other subdirectories, such as IMP, Lambda and Auth.

4.9 The examples directories

Directory HOL/Auth contains theories for proving the correctness of cryptographic protocols. The approach is based upon operational semantics [38] rather than the more usual belief logics. On the same directory are proofs for some standard examples, such as the Needham-Schroeder public-key authentication protocol [36] and the Otway-Rees protocol.

Directory HOL/IMP contains a formalization of various denotational, operational and axiomatic semantics of a simple while-language, the necessary equivalence proofs, soundness and completeness of the Hoare rules with respect to the denotational semantics, and soundness and completeness of a verification condition generator. Much of development is taken from Winskel [50]. For details see [27].

Directory `HOL/Hoare` contains a user friendly surface syntax for Hoare logic, including a tactic for generating verification-conditions.

Directory `HOL/MiniML` contains a formalization of the type system of the core functional language Mini-ML and a correctness proof for its type inference algorithm $\mathcal{W}$ [23, 25].

Directory `HOL/Lambda` contains a formalization of untyped λ-calculus in de Bruijn notation and Church-Rosser proofs for β and η reduction [26].

Directory `HOL/Subst` contains Martin Coen's mechanization of a theory of substitutions and unifiers. It is based on Paulson's previous mechanisation in `LCF` [31] of Manna and Waldinger's theory [21]. It demonstrates a complicated use of `recdef`, with nested recursion.

Directory `HOL/Induct` presents simple examples of (co)inductive definitions.

- Theory `PropLog` proves the soundness and completeness of classical propositional logic, given a truth table semantics. The only connective is $\rightarrow$. A Hilbert-style axiom system is specified, and its set of theorems defined inductively. A similar proof in `ZF` is described elsewhere [35].

- Theory `Term` develops an experimental recursive type definition; the recursion goes through the type constructor *list*.

- Theory `Simult` constructs mutually recursive sets of trees and forests, including induction and recursion rules.

- The definition of lazy lists demonstrates methods for handling infinite data structures and coinduction in higher-order logic [37].[7] Theory `LList` defines an operator for corecursion on lazy lists, which is used to define a few simple functions such as map and append. A coinduction principle is defined for proving equations on lazy lists.

- Theory `LFilter` defines the filter functional for lazy lists. This functional is notoriously difficult to define because finding the next element meeting the predicate requires possibly unlimited search. It is not computable, but can be expressed using a combination of induction and corecursion.

- Theory `Exp` illustrates the use of iterated inductive definitions to express a programming language semantics that appears to require mutual induction. Iterated induction allows greater modularity.

Directory `HOL/ex` contains other examples and experimental proofs in `HOL`.

- Theory `Recdef` presents many examples of using `recdef` to define recursive functions. Another example is `Fib`, which defines the Fibonacci function.

[7]To be precise, these lists are *potentially infinite* rather than lazy. Lazy implies a particular operational semantics.

- Theory `Primes` defines the Greatest Common Divisor of two natural numbers and proves a key lemma of the Fundamental Theorem of Arithmetic: if p is prime and p divides $m \times n$ then p divides m or p divides n.
- Theory `Primrec` develops some computation theory. It inductively defines the set of primitive recursive functions and presents a proof that Ackermann's function is not primitive recursive.
- File `cla.ML` demonstrates the classical reasoner on over sixty predicate calculus theorems, ranging from simple tautologies to moderately difficult problems involving equality and quantifiers.
- File `meson.ML` contains an experimental implementation of the MESON proof procedure, inspired by Plaisted [42]. It is much more powerful than Isabelle's classical reasoner. But it is less useful in practice because it works only for pure logic; it does not accept derived rules for the set theory primitives, for example.
- File `mesontest.ML` contains test data for the MESON proof procedure. These are mostly taken from Pelletier [41].
- File `set.ML` proves Cantor's Theorem, which is presented in §4.10 below, and the Schröder-Bernstein Theorem.
- Theory `MT` contains Jacob Frost's formalization [14] of Milner and Tofte's coinduction example [24]. This substantial proof concerns the soundness of a type system for a simple functional language. The semantics of recursion is given by a cyclic environment, which makes a coinductive argument appropriate.

4.10 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:

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

Viewing types as sets, $\alpha \Rightarrow \mathit{bool}$ 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 $\alpha\ \mathit{set}$ and the operator `range`. 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 Set.thy "?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 $?S \in \texttt{range}\, f$ 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 `set_current_thy "Set"`. Otherwise the default claset may not contain the rules for set theory.

Chapter 5

First-Order Sequent Calculus

The theory `LK` implements classical first-order logic through Gentzen's sequent calculus (see Gallier [15] or Takeuti [46]). 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.

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`.

No generic packages are instantiated, since Isabelle does not provide packages for sequent calculi at present. `LK` implements a classical logic theorem prover that is as powerful as the generic classical reasoner, except that it does not perform equality reasoning.

5.1 Unification for lists

Higher-order unification includes associative unification as a special case, by an encoding that involves function composition [18, 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:

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

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 ($\to$)
`<->`	$[o, o] \Rightarrow o$	Right 25	biconditional ($\leftrightarrow$)

Infixes

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

Translations

Figure 5.1: Syntax of `LK`

```
    prop  =  sequence |- sequence

sequence  =  elem  (, elem)*
          |  empty

    elem  =  $ id
          |  $ var
          |  formula

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

Figure 5.2: Grammar of LK

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 [13].

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.

5.2 Syntax and rules of inference

Figure 5.1 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. 5.3) does not entail the Axiom of Choice because it requires uniqueness.

```
basic       $H, P, $G |- $E, P, $F
thinR       $H |- $E, $F ==> $H |- $E, P, $F
thinL       $H, $G |- $E ==> $H, P, $G |- $E
cut         [| $H |- $E, P;  $H, P |- $E |] ==> $H |- $E
```

Structural rules

```
refl        $H |- $E, a=a, $F
sym         $H |- $E, a=b, $F ==> $H |- $E, b=a, $F
trans       [| $H|- $E, a=b, $F;  $H|- $E, b=c, $F |] ==>
            $H|- $E, a=c, $F
```

Equality rules

```
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 5.3: Rules of LK

```
conR        $H |- $E, P, $F, P ==> $H |- $E, P, $F
conL        $H, P, $G, P |- $E ==> $H, P, $G |- $E

symL        $H, $G, B = A |- $E ==> $H, A = B, $G |- $E

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
```

Figure 5.4: Derived rules for `LK`

Figure 5.2 presents the grammar of `LK`. Traditionally, Γ and Δ are meta-variables for sequences. In Isabelle's notation, the prefix `$` on a variable makes it range over sequences. In a sequent, anything not prefixed by `$` is taken as a formula.

Figure 5.3 presents 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 — they do not require exchange or contraction rules. The contraction rules are actually derivable (via cut) in this formulation.

Figure 5.4 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/LK` in the sources for complete listings of the rules and derived rules.

5.3 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 can not 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`.

5.4 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`.

5.5 Packaging sequent rules

Section 1.2 described the distinction between safe and unsafe rules. An unsafe rule may reduce a provable goal to an unprovable set of subgoals, and should only be used as a last resort. Typical examples are the weakened quantifier rules `allL_thin` and `exR_thin`.

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:

```
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
```

`empty_pack` is the empty pack.

`prop_pack` contains the propositional rules, namely those for $\land$, $\lor$, $\neg$, $\rightarrow$ 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*.

5.6 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.

5.6.1 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*.

5.6.2 Method B

```
safe_goal_tac : pack -> int -> tactic
step_tac      : pack -> int -> tactic
fast_tac      : pack -> int -> tactic
best_tac      : pack -> int -> tactic
```

These tactics are precisely 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`).

5.7 A simple example of classical reasoning

The theorem $\vdash \exists y . \forall x . P(y) \rightarrow P(x)$ is a standard example of the classical treatment of the existential quantifier. Classical reasoning is easy using `LK`, as you can see by comparing this proof with the one given in §2.7. 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 LK.thy "|- 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 LK.thy "|- 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!
```

5.8 A more complex proof

Many of Pelletier's test problems for theorem provers [41] 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 has a short manual proof. See the directory LK/ex for many more examples.

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

```
goal LK.thy "|- ~ (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!
```

Chapter 6

Constructive Type Theory

Martin-Löf's Constructive Type Theory [22, 29] 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 [47] gives a readable and thorough account of Type Theory. Nuprl is an elaborate implementation [8]. ALF is a more recent tool that allows proof terms to be edited directly [20].

Isabelle's original formulation of Type Theory was a kind of sequent calculus, following Martin-Löf [22]. 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 $\Longrightarrow$:

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

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

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

To justify the CTT formulation it is probably best to appeal directly to the semantic explanations of the rules [22], 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.

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

name	*meta-type*	*description*
`Type`	$t \to \mathit{prop}$	judgement form
`Eqtype`	$[t, t] \to \mathit{prop}$	judgement form
`Elem`	$[i, t] \to \mathit{prop}$	judgement form
`Eqelem`	$[i, i, t] \to \mathit{prop}$	judgement form
`Reduce`	$[i, i] \to \mathit{prop}$	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 6.1: The constants of `CTT`

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.

6.1 Syntax

The constants are shown in Fig. 6.1. 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. 6.2) is based on that of Nordström et al. [29]. 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 using general 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.

6.2 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.

[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

```
prop  =  type type
      |  type = type
      |  term : type
      |  term = term : type

type  =  expression of type t
      |  PROD id : type . type
      |  SUM  id : type . type

term  =  expression of type i
      |  lam id id* . term
      |  < term , term >
```

Grammar

Figure 6.2: 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 6.3: General equality rules

Figure 6.3 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 6.4 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 axiom and cannot be derived without universes [22, 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 [47, page 104].

Figure 6.5 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.

```
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 6.4: 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 6.5: 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 6.6: Rules for the sum type $\sum_{x\in A} B[x]$

Figure 6.6 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 6.7 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 6.8 shows the rules for the empty and unit types, F and T. They correspond to the predicate calculus rules for absurdity and truth.

Figure 6.9 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 [29].

Figure 6.10 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 6.12 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

```
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 6.7: 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 6.8: 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 6.9: 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 6.10: Derived rules for `CTT`

corresponding bound variables in the premises.

6.3 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`.

6.4 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.

6.5 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 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

[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.

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.

6.6 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, $\vee$-elimination with the assumption $P \vee Q$ creates one subgoal assuming P and another assuming Q, and $P \vee Q$ 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`.

6.7 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 6.11 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 $\mathit{succ}(v) = b$.

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

6.8 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.

`CTT/ex/elim.ML` contains some examples from Martin-Löf [22], 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.

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 6.11: The theory of arithmetic

6.9 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 CTT.thy "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 [NI0] 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 CTT.thy "lam n. rec(n, 0, %x y.tt) : ?A";
```

6.10 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:

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

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

```
val prems = goal CTT.thy
    "[| 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 [8]. 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.

6.11 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 CTT.thy
    "[| 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 [22, page 58].

6.12 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:

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

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. 6.10) and the type checking tactics, we can prove the theorem in nine steps.

```
val prems = goal CTT.thy
    "[| 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 [22, page 50].

Bibliography

[1] J. R. Abrial and G. Laffitte. Towards the mechanization of the proofs of some classical theorems of set theory. preprint, February 1993.

[2] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Academic Press, 1986.

[3] David Basin and Matt Kaufmann. The Boyer-Moore prover and Nuprl: An experimental comparison. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 89–119. Cambridge University Press, 1991.

[4] Robert Boyer, Ewing Lusk, William McCune, Ross Overbeek, Mark Stickel, and Lawrence Wos. Set theory in first-order logic: Clauses for Gödel's axioms. *Journal of Automated Reasoning*, 2(3):287–327, 1986.

[5] J. Camilleri and T. F. Melham. Reasoning with inductively defined relations in the HOL theorem prover. Technical Report 265, Computer Laboratory, University of Cambridge, August 1992.

[6] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

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

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

[9] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, 1990.

[10] Keith J. Devlin. *Fundamentals of Contemporary Set Theory.* Springer, 1979.

[11] Michael Dummett. *Elements of Intuitionism.* Oxford University Press, 1977.

[12] Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic*, 57(3):795–807, 1992.

[13] 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.

[14] Jacob Frost. A case study of co-induction in Isabelle HOL. Technical Report 308, Computer Laboratory, University of Cambridge, August 1993.

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

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

[17] Paul R. Halmos. *Naive Set Theory*. Van Nostrand, 1960.

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

[19] Kenneth Kunen. *Set Theory: An Introduction to Independence Proofs*. North-Holland, 1980.

[20] 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.

[21] Zohar Manna and Richard Waldinger. Deductive synthesis of the unification algorithm. *Science of Computer Programming*, 1(1):5–48, 1981.

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

[23] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[24] Robin Milner and Mads Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87:209–220, 1991.

[25] Dieter Nazareth and Tobias Nipkow. Formal verification of algorithm W: The monomorphic case. In von Wright et al. [48], pages 331–345.

[26] Tobias Nipkow. More Church-Rosser proofs (in Isabelle/HOL). In Michael McRobbie and John K. Slaney, editors, *Automated Deduction — CADE-13 International Conference*, LNAI 1104, pages 733–747. Springer, 1996.

[27] Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. In V. Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *LNCS*, pages 180–192. Springer, 1996.

[28] Philippe Noël. Experimenting with Isabelle in ZF set theory. *Journal of Automated Reasoning*, 10(1):15–58, 1993.

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

[30] Christine Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. Research Report 92-49, LIP, Ecole Normale Supérieure de Lyon, December 1992.

[31] Lawrence C. Paulson. Verifying the unification algorithm in LCF. *Science of Computer Programming*, 5:143–170, 1985.

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

[33] Lawrence C. Paulson. Set theory for verification: I. From foundations to functions. *Journal of Automated Reasoning*, 11(3):353–389, 1993.

[34] Lawrence C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In Alan Bundy, editor, *Automated Deduction — CADE-12 International Conference*, LNAI 814, pages 148–161. Springer, 1994.

[35] Lawrence C. Paulson. Set theory for verification: II. Induction and recursion. *Journal of Automated Reasoning*, 15(2):167–215, 1995.

[36] Lawrence C. Paulson. Mechanized proofs of security protocols: Needham-Schroeder with public keys. Technical Report 413, Computer Laboratory, University of Cambridge, January 1997.

[37] Lawrence C. Paulson. Mechanizing coinduction and corecursion in higher-order logic. *Journal of Logic and Computation*, 7(2):175–204, March 1997.

[38] Lawrence C. Paulson. Proving properties of security protocols by induction. In *10th Computer Security Foundations Workshop*, pages 70–83. IEEE Computer Society Press, 1997.

[39] 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.

[40] Lawrence C. Paulson. A concrete final coalgebra theorem for ZF set theory. In Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for Proofs and Programs: International Workshop TYPES '94*, LNCS 996, pages 120–139. Springer, published 1995.

[41] 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.

[42] David A. Plaisted. A sequent-style model elimination strategy and a positive refinement. *Journal of Automated Reasoning*, 6(4):389–402, 1990.

[43] Art Quaife. Automated deduction in von Neumann-Bernays-Gödel set theory. *Journal of Automated Reasoning*, 8(1):91–147, 1992.

[44] Konrad Slind. Function definition in higher-order logic. In von Wright et al. [48].

[45] Patrick Suppes. *Axiomatic Set Theory.* Dover, 1972.

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

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

[48] J. von Wright, J. Grundy, and J. Harrison, editors. *Theorem Proving in Higher Order Logics: TPHOLs '96*, LNCS 1125, 1996.

[49] A. N. Whitehead and B. Russell. *Principia Mathematica.* Cambridge University Press, 1962. Paperback edition to *56, abridged from the 2nd edition (1927).

[50] Glynn Winskel. *The Formal Semantics of Programming Languages.* MIT Press, 1993.

Index