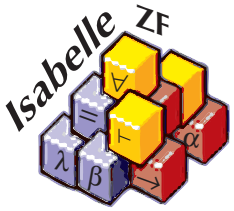


Isabelle's Logics: FOL and ZF

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

With Contributions by Tobias Nipkow and Markus Wenzel

August 15, 2018

Abstract

This manual describes Isabelle's formalizations of many-sorted first-order logic (`FOL`) and Zermelo-Fraenkel set theory (`ZF`). See the *Reference Manual* for general Isabelle commands, and *Introduction to Isabelle* for an overall tutorial.

This manual is part of the earlier Isabelle documentation, which is somewhat superseded by the Isabelle/HOL *Tutorial* [11]. However, the present document is the only available documentation for Isabelle's versions of first-order logic and set theory. Much of it is concerned with the primitives for conducting proofs using the ML top level. It has been rewritten to use the Isar proof language, but evidence of the old ML orientation remains.

Acknowledgements

Markus Wenzel made numerous improvements. Philippe de Groote contributed to ZF. Philippe Noël and Martin Coen made many contributions to ZF. The research has been funded by the EPSRC (grants GR/G53279, GR/H40570, GR/K57381, GR/K77051, GR/M75440) and by ESPRIT (projects 3245: Logical Frameworks, and 6453: Types) and by the DFG Schwerpunktprogramm *Deduktion*.

Contents

Chapter 1

Syntax definitions

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

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

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

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

A **binder** is a symbol associated with a constant of type $(\sigma \Rightarrow \tau) \Rightarrow \tau'$. For instance, we may declare $\forall$ as a binder for the constant *All*, which has type $(\alpha \Rightarrow o) \Rightarrow o$. This defines the syntax $\forall x\,.\,t$ to mean $All(\lambda x\,.\,t)$. We can also write $\forall x_1 \ldots x_m\,.\,t$ to abbreviate $\forall x_1 \ldots .\, \forall x_m\,.\,t$; this is possible for any

constant provided that τ and τ' are the same type. The Hilbert description operator $\varepsilon x \,.\, P\,x$ has type $(\alpha \Rightarrow \mathit{bool}) \Rightarrow \alpha$ and normally binds only one variable. ZF's bounded quantifier $\forall x \in A \,.\, P(x)$ cannot be declared as a binder because it has type $[i, i \Rightarrow o] \Rightarrow o$. The syntax for binders allows type constraints on bound variables, as in

$$\forall (x{::}\alpha)\ (y{::}\beta)\ z{::}\gamma \,.\, Q(x, y, z)$$

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

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

Chapter 2

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 names. Negation is defined in the usual way for intuitionistic logic; $\neg P$ abbreviates $P \to \bot$. The biconditional ($\leftrightarrow$) is defined through $\wedge$ 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 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 ($\wedge E$), ($\to E$), 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 Sect. 2.3).

See the on-line theory library 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, which is invoked through the method `simp`. Both equality ($=$) and the biconditional ($\leftrightarrow$) may be used for rewriting.
- It instantiates the classical reasoner, which is invoked mainly through the methods `blast` and `auto`. See Sect. 2.4 for details.

! Simplifying $a = b \wedge P(a)$ to $a = b \wedge P(b)$ is often 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 the theorem `conj_cong` as a congruence rule in simplification, `simp cong: 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 $\neg P \vee Q$. 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 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: $(P \wedge Q) \to S$ is equivalent to $P \to (Q \to S)$, and $(P \vee Q) \to S$ is equivalent to the conjunction of $P \to S$ and $Q \to S$. The other `..._impE`

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

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 [8]. However, the tactics given below are not complete for first-order logic because they discard universally quantified assumptions after a single use. These are ML functions, and are listed below with their ML type:

```
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 ML structure `IntPr` and resemble the tactics of Isabelle's classical reasoner. There are no corresponding Isar methods, but you can use the *tactic* method to call ML tactics in an Isar script:

apply *(tactic * IntPr.fast_tac 1*)*

Here is a description of each tactic:

`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.

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

`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) [24].

```
~~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

$$\dfrac{\begin{array}{c}[\neg P]\\ \vdots \\ P\end{array}}{P} \qquad (\textit{classical})$$

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

The classical reasoner is installed. The most useful methods are *blast* (pure classical reasoning) and *auto* (classical reasoning combined with simplification), but the full range of methods is available, including *clarify*, *fast*, *best* and *force*. See the the *Reference Manual* and the *Tutorial* [11] for more discussion of classical proof methods.

2.5 An intuitionistic example

Here is a session similar to one in the book *Logic and Computation* [15, pages 222–3]. It illustrates the treatment of quantifier reasoning, which is different in Isabelle than it is in LCF-based theorem provers such as HOL.

The theory header specifies that we are working in intuitionistic logic by designating *IFOL* rather than *FOL* as the parent theory:

theory *IFOL_examples* **imports** *IFOL*
begin

The proof begins by entering the goal, then applying the rule $(\to I)$.

lemma *"(EX y. ALL x. Q(x,y)) --> (ALL x. EX y. Q(x,y))"*
1. $(\exists y.\ \forall x.\ Q(x, y)) \longrightarrow (\forall x.\ \exists y.\ Q(x, y))$
apply *(rule impI)*
1. $\exists y.\ \forall x.\ Q(x, y) \Longrightarrow \forall x.\ \exists y.\ Q(x, y)$

Isabelle's output is shown as it would appear using Proof General. In this example, we shall never have more than one subgoal. Applying $(\to I)$ replaces $\longrightarrow$ by $\Longrightarrow$, so that $\exists y\,.\forall x\,.\,Q(x,y)$ becomes an assumption. We have the choice of $(\exists E)$ and $(\forall I)$; let us try the latter.

apply *(rule allI)*
1. $\bigwedge x.\ \exists y.\ \forall x.\ Q(x, y) \Longrightarrow \exists y.\ Q(x, y)$ $(*)$

Applying $(\forall I)$ replaces the $\forall x$ (in ASCII, *ALL x*) by $\bigwedge x$ (or *!!x*), replacing FOL's universal quantifier by Isabelle's meta universal quantifier. 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?

apply *(rule exI)*
1. $\bigwedge x.\ \exists y.\ \forall x.\ Q(x, y) \Longrightarrow Q(x, ?y2(x))$

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:

apply *(erule exE)*
1. $\bigwedge x\ y.\ \forall x.\ Q(x, y) \Longrightarrow Q(x, ?y2(x))$

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 Proof General, we can return to the critical point, marked $(*)$ above. This time we apply $(\exists E)$:

apply *(erule exE)*
1. $\bigwedge x\ y.\ \forall x.\ Q(x, y) \Longrightarrow \exists y.\ Q(x, y)$

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.

```
apply (rule exI)
 1. ⋀x y. ∀x. Q(x, y) ⟹ Q(x, ?y3(x, y))
apply (erule allE)
 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`.

```
apply assumption
No subgoals!
done
```

The theorem was proved in six method invocations, not counting the abandoned ones. But proof checking is tedious, and the ML tactic `IntPr.fast_tac` can prove the theorem in one step.

```
lemma "(EX y. ALL x. Q(x,y)) -->  (ALL x. EX y. Q(x,y))"
 1. (∃y. ∀x. Q(x, y)) ⟶ (∀x. ∃y. Q(x, y))
by (tactic {*IntPr.fast_tac 1*})
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 [7, 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 $(P \to Q) \vee (Q \to P)$. The first step is apply the `unfold` method, which expands negations to implications using the definition $\neg P \equiv P \to \bot$ and allows use of the special implication rules.

```
lemma "~ ~ ((P-->Q) | (Q-->P))"
 1. ¬ ¬ ((P ⟶ Q) ∨ (Q ⟶ P))
apply (unfold not_def)
 1. ((P ⟶ Q) ∨ (Q ⟶ P) ⟶ False) ⟶ False
```

The next step is a trivial use of $(\to I)$.

[1] This remark holds only for propositional logic, not if P is allowed to contain quantifiers.

apply *(rule impI)*
1. (P ⟶ Q) ∨ (Q ⟶ P) ⟶ False ⟹ False

By $(\to E)$ it would suffice to prove $(P \to Q) \vee (Q \to P)$, 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.

apply *(erule disj_impE)*
1. ⟦(P ⟶ Q) ⟶ False; (Q ⟶ P) ⟶ False⟧ ⟹ False

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

apply *(erule imp_impE)*
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 $Q \to P$; applying *imp_impE* is simpler.

apply *(erule imp_impE)*
1. ⟦P; Q ⟶ False; Q; P ⟶ False⟧ ⟹ P
2. ⟦P; Q ⟶ False; False⟧ ⟹ Q
3. ⟦(Q ⟶ P) ⟶ False; False⟧ ⟹ False

The three subgoals are all trivial.

apply *assumption*
1. ⟦P; Q ⟶ False; False⟧ ⟹ Q
2. ⟦(Q ⟶ P) ⟶ False; False⟧ ⟹ False
apply *(erule FalseE)+*
No subgoals!
done

This proof is also trivial for the ML tactic *IntPr.fast_tac*.

2.7 A classical example

To illustrate classical logic, we shall prove the theorem $\exists y. \forall x. P(y) \to P(x)$. 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. First, we must work in a theory based on classical logic, the theory *FOL*:

theory *FOL_examples* **imports** *FOL*
begin

The formal proof does not conform in any obvious way to the sketch given above. Its key step is its first rule, `exCI`, a classical version of $(\exists I)$ that allows multiple instantiation of the quantifier.

```
lemma "EX y. ALL x. P(y)-->P(x)"
 1. ∃y. ∀x. P(y) ⟶ P(x)
apply (rule exCI)
 1. ∀y. ¬ (∀x. P(y) ⟶ P(x)) ⟹ ∀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.

```
apply (rule allI)
 1. ⋀x. ∀y. ¬ (∀x. P(y) ⟶ P(x)) ⟹ P(?a) ⟶ P(x)
apply (rule impI)
 1. ⋀x. ⟦∀y. ¬ (∀x. P(y) ⟶ P(x)); P(?a)⟧ ⟹ P(x)
```

By the duality between $\exists$ and $\forall$, applying $(\forall E)$ is equivalent to applying $(\exists I)$ again.

```
apply (erule allE)
 1. ⋀x. ⟦P(?a); ¬ (∀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 $(\forall I)$ and apply it using `erule`.

```
apply (erule allI [THEN [2] swap])
 1. ⋀x xa. ⟦P(?a); ¬ P(x)⟧ ⟹ P(?y3(x)) ⟶ P(xa)
```

The operand of `erule` above denotes the following theorem:

```
⟦¬ (∀x. ?P1(x)); ⋀x. ¬ ?P ⟹ ?P1(x)⟧ ⟹ ?P
```

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

```
apply (rule impI)
 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`.

```
apply (erule notE)
 1. ⋀x xa. ⟦P(?a); P(?y3(x))⟧ ⟹ P(x)
apply assumption
No subgoals!
done
```

The civilised way to prove this theorem is using the `blast` method, which automatically uses the classical form of the rule $(\exists I)$:

lemma *"EX y. ALL x. P(y)-->P(x)"*
1. ∃y. ∀x. P(y) ⟶ P(x)
by *blast*
No subgoals!

If this theorem seems counterintuitive, then perhaps you are an intuitionist. In constructive logic, proving $\exists y \,.\, \forall x \,.\, P(y) \to P(x)$ requires exhibiting a particular term t such that $\forall x \,.\, P(t) \to P(x)$, 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 $if(P, Q, R)$, where

$$if(P, Q, R) \equiv P \wedge Q \vee \neg P \wedge R. \qquad (if)$$

Theorems about *if* can be proved by treating this as an abbreviation, replacing $if(P, Q, R)$ by $P \wedge Q \vee \neg P \wedge R$ 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 $if(P, Q, R)$ directly, without reference to its definition. The simple identity

$$if(P, Q, R) \leftrightarrow (P \to Q) \wedge (\neg P \to R)$$

suggests that the *if*-introduction rule should be

$$\frac{\begin{array}{cc}[P] & [\neg P] \\ \vdots & \vdots \\ Q & R\end{array}}{if(P, Q, R)}\ (if\ I)$$

The *if*-elimination rule reflects the definition of $if(P, Q, R)$ and the elimination rules for $\vee$ and $\wedge$.

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

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

theory *If* **imports** *FOL*
begin
constdefs
if :: "[o,o,o]=>o"
"if(P,Q,R) == P&Q | ~P&R"

We create the file `If.thy` containing these declarations. (This file is on directory `FOL/ex` in the Isabelle distribution.) Typing

```
use_thy "If";
```

loads that theory and sets it to be the current context.

2.8.1 Deriving the introduction rule

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

The introduction rule, given the premises $P \Longrightarrow Q$ and $\neg P \Longrightarrow R$, concludes $if(P, Q, R)$. We propose this lemma and immediately simplify using `if_def` to expand the definition of `if` in the subgoal.

```
lemma ifI: "[| P ==> Q; ~P ==> R |] ==> if(P,Q,R)"
 1. ⟦P ⟹ Q; ¬ P ⟹ R⟧ ⟹ if(P, Q, R)
apply (simp add: if_def)
 1. ⟦P ⟹ Q; ¬ P ⟹ R⟧ ⟹ P ∧ Q ∨ ¬ P ∧ R
```

The rule's premises, although expressed using meta-level implication ($\Longrightarrow$) are passed as ordinary implications to `blast`.

```
apply blast
No subgoals!
done
```

2.8.2 Deriving the elimination rule

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

```
lemma ifE:
   "[| if(P,Q,R);  [|P; Q|] ==> S; [|~P; R|] ==> S |] ==> S"
 1. ⟦if(P, Q, R); ⟦P; Q⟧ ⟹ S; ⟦¬ P; R⟧ ⟹ S⟧ ⟹ S
apply (simp add: if_def)
 1. ⟦P ∧ Q ∨ ¬ P ∧ R; ⟦P; Q⟧ ⟹ S; ⟦¬ P; R⟧ ⟹ S⟧ ⟹ S
```

The proof script is the same as before: `simp` followed by `blast`:

```
apply blast
No subgoals!
done
```

2.8.3 Using the derived rules

Our new derived rules, *ifI* and *ifE*, permit natural proofs of theorems such as the following:

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

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.

```
lemma if_commute:
    "if(P, if(Q,A,B), if(Q,C,D)) <-> if(Q, if(P,A,C), if(P,B,D))"
apply (rule iffI)
 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.

```
apply (erule ifE)
 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))
apply (erule ifE)
 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.

```
apply (rule ifI)
 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))
apply (rule ifI)
 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*.

```
declare ifI [intro!]
declare ifE [elim!]
```

With these declarations, we could have proved this theorem with a single call to `blast`. Here is another example:

```
lemma "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
```

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:

```
lemma "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, we simply unfold using the definition of *if*:

```
apply (simp add: if_def)
 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, and it falls to `blast`:

```
apply blast
No subgoals!
```

Expanding definitions reduces the extended logic to the base logic. This approach has its merits, but it can be slow. In these examples, proofs using the derived rules for `if` run about six times faster than proofs using just the rules of first-order logic.

Expanding definitions can also make it harder to diagnose errors. 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:

```
lemma "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))
```

Calling `blast` yields an uninformative failure message. We can get a closer look at the situation by applying `auto`.

apply *auto*
1. $[\![A;\ \neg\ P;\ R]\!] \Longrightarrow B$
2. $[\![B;\ \neg\ P;\ \neg\ R]\!] \Longrightarrow A$
3. $[\![B;\ \neg\ P;\ R]\!] \Longrightarrow A$
4. $[\![\neg\ R;\ A;\ \neg\ B;\ \neg\ P]\!] \Longrightarrow False$

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 first-order logic:

lemma *"if(if(P,Q,R), A, B) <-> if(P, if(Q,A,B), if(R,B,A))"*
1. $if(if(P,\ Q,\ R),\ A,\ B) \longleftrightarrow if(P,\ if(Q,\ A,\ B),\ if(R,\ B,\ A))$

apply *(simp add: if_def)*
1. $(P \wedge Q \vee \neg\ P \wedge R) \wedge A \vee (\neg\ P \vee \neg\ Q) \wedge (P \vee \neg\ R) \wedge B \longleftrightarrow$
   $P \wedge (Q \wedge A \vee \neg\ Q \wedge B) \vee \neg\ P \wedge (R \wedge B \vee \neg\ R \wedge A)$

Again `blast` would fail, so we try `auto`:

apply *(auto)*
1. $[\![A;\ \neg\ P;\ R]\!] \Longrightarrow B$
2. $[\![A;\ \neg\ P;\ R;\ \neg\ B]\!] \Longrightarrow Q$
3. $[\![B;\ \neg\ R;\ \neg\ P;\ \neg\ A]\!] \Longrightarrow False$
4. $[\![B;\ \neg\ P;\ \neg\ A;\ \neg\ R;\ Q]\!] \Longrightarrow False$
5. $[\![B;\ \neg\ Q;\ \neg\ R;\ \neg\ P;\ \neg\ A]\!] \Longrightarrow False$
6. $[\![B;\ \neg\ A;\ \neg\ P;\ R]\!] \Longrightarrow False$
7. $[\![\neg\ P;\ A;\ \neg\ B;\ \neg\ R]\!] \Longrightarrow False$
8. $[\![\neg\ P;\ A;\ \neg\ B;\ \neg\ R]\!] \Longrightarrow Q$

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 your definitions usually makes proofs more difficult. 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 [9, 23] 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. Some of it is based on the work of Noël [12].

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. `ZF` provides both the integers and the natural numbers. 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.

`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. It provides a streamlined syntax for defining primitive recursive functions over datatypes.

Published articles [16, 18] 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 [20].

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 [3, 22]. 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 col-

lections 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 $\neg(a \in b)$, which is equivalent to $a \notin b$. 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 `cons`:

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

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`,`λx`.` $P[x]$`)`	separation
`{`y`.` x`:`A`,` $Q[x,y]$`}`	`Replace(`A`,`$\lambda x\ y$`.` $Q[x,y]$`)`	replacement
`{`$b[x]$`.` x`:`A`}`	`RepFun(`A`,`λx`.` $b[x]$`)`	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`,`λx`.` $B[x]$`)`	general product
SUM x`:`A`.` $B[x]$	`Sigma(`A`,`λx`.` $B[x]$`)`	general sum
A `->` B	`Pi(`A`,`λx`.` B`)`	function space
A `*` B	`Sigma(`A`,`λx`.` B`)`	binary product
THE x`.` $P[x]$	`The(`λx`.` $P[x]$`)`	definite description
lam x`:`A`.` $b[x]$	`Lambda(`A`,`λx`.` $b[x]$`)`	λ-abstraction
ALL x`:`A`.` $P[x]$	`Ball(`A`,`λx`.` $P[x]$`)`	bounded $\forall$
EX x`:`A`.` $P[x]$	`Bex(`A`,`λx`.` $P[x]$`)`	bounded $\exists$

Figure 3.2: Translations for ZF

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

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

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 occurrences of x. It abbreviates the set *Collect(A,*λx *.* $P[x]$*)*, 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

term	=	expression of type i
	|	`let` *id* `=` *term* `;` ... `;` *id* `=` *term* `in` *term*
	|	`if` *term* `then` *term* `else` *term*
	|	`{` *term* (`,`*term*)* `}`
	|	`<` *term* (`,`*term*)* `>`
	|	`{` *id*`:`*term* `.` *formula* `}`
	|	`{` *id* `.` *id*`:`*term*`,` *formula* `}`
	|	`{` *term* `.` *id*`:`*term* `}`
	|	*term* ‘‘ *term*
	|	*term* `-`‘‘ *term*
	|	*term* ‘ *term*
	|	*term* `*` *term*
	|	*term* ∩*term*
	|	*term* ∪*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

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 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**:

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

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`. `$P[x]$.

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

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

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
extension:        A = B  <->  A ⊆ B & B ⊆ A

Union_iff:        A ∈ Union(C) <-> (∃B ∈ C. A ∈ B)
Pow_iff:          A ∈ Pow(B) <-> A ⊆ B
foundation:       A=0 | (∃x ∈ A. ∀y ∈ x. y ∉ A)

replacement:      (∀x ∈ A. ∀y z. P(x,y) & P(x,z) --> y=z) ==>
                  b ∈ PrimReplace(A,P) <-> (∃x∈A. P(x,b))
```

The Zermelo-Fraenkel Axioms

```
Replace_def: Replace(A,P) ==
                  PrimReplace(A, %x y. (∃!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) . ∀y ∈ A. x ∈ y}
Un_def:      A ∪ B  == Union(Upair(A,B))
Int_def:     A ∩ 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) ∪ A
succ_def:    succ(i) == cons(i,i)
infinity:    0 ∈ Inf & (∀y ∈ Inf. succ(y) ∈ Inf)
```

Finite and infinite sets

```
Pair_def:      <a,b>      == {{a,a}, {a,b}}
split_def:     split(c,p) == THE y. ∃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) == ⋃x ∈ A. ⋃y ∈ B(x). {<x,y>}
```

Ordered pairs and Cartesian products

```
converse_def: converse(r) == {z. w∈r, ∃x y. w=<x,y> & z=<y,x>}
domain_def:   domain(r)   == {x. w ∈ r, ∃y. w=<x,y>}
range_def:    range(r)    == domain(converse(r))
field_def:    field(r)    == domain(r) ∪ range(r)
image_def:    r `` A      == {y∈range(r) . ∃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)). ∀x∈A. ∃!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

3.4 The Zermelo-Fraenkel axioms

The axioms appear in Fig. 3.4. They resemble those presented by Suppes [23]. 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.

The traditional replacement axiom asserts

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

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

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

Thus $y \in \mathtt{Replace}(A, P)$ 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 replacement for meta-level functions (`RepFun`) and definite descriptions (`The`). Axioms for separation (`Collect`) and unordered pairs (`Upair`) are traditionally assumed, but they actually follow from replacement [23, 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.

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.

```
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

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

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

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

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.

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.

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

```
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); ∀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∈(⋃x∈A. B(x))
UN_E:   [| b∈(⋃x∈A. B(x));  !!x.[| x∈A;  b∈B(x) |] ==> R
        |] ==> R

INT_I:  [| !!x. x∈A ==> b∈B(x);  a∈A |] ==> b∈(⋂x∈A. B(x))
INT_E:  [| b∈(⋂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 ∪ B
UnI2:      c∈B ==> c∈A ∪ B
UnCI:      (c ∉ B ==> c∈A) ==> c∈A ∪ B
UnE:       [| c∈A ∪ B;  c∈A ==> P;  c∈B ==> P |] ==> P

IntI:      [| c∈A;  c∈B |] ==> c∈A ∩ B
IntD1:     c∈A ∩ B ==> c∈A
IntD2:     c∈A ∩ B ==> c∈B
IntE:      [| c∈A ∩ 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:         ∃! x. P(x) ==> P(THE x. P(x))

if_P:          P ==> (if P then a else b) = a
if_not_P:     ~P ==> (if P then a else 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 ∪ B
Un_upper2:      B ⊆ A ∪ B
Un_least:       [| A ⊆ C;  B ⊆ C |] ==> A ∪ B ⊆ C

Int_lower1:     A ∩ B ⊆ A
Int_lower2:     A ∩ B ⊆ B
Int_greatest:   [| C ⊆ A;  C ⊆ B |] ==> C ⊆ A ∩ B

Diff_subset:    A-B ⊆ A
Diff_contains:  [| C ⊆ A;  C ∩ B = 0 |] ==> C ⊆ A-B

Collect_subset: Collect(A,P) ⊆ A
```

Figure 3.14: Subset and lattice properties

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.

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 corresponding rules. A few other laws involving subsets are included. Reasoning directly about subsets often yields clearer proofs than reasoning about the membership relation. Section 3.13 below presents an example of this, proving the equation $\mathit{Pow}(A) \cap \mathit{Pow}(B) = \mathit{Pow}(A \cap B)$.

3.5.4 Ordered pairs

Figure 3.15 presents the rules governing ordered pairs, projections and general sums — in particular, that $\{\{a\}, \{a, b\}\}$ functions as an ordered

```
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

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 `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 $\langle a, b\rangle \in$ `Sigma`(A, B) implies $a \in A$ and $b \in B(a)$.

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

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

Nested patterns are translated recursively: `%<`x`,`y`,`z`>.` $t \rightsquigarrow$ `%<`x`,<`y`,`z`>>.` $t \rightsquigarrow$ `split(%`x`.%<`y`,`z`>.` t`)` $\rightsquigarrow$ `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 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. Here are some important examples:

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

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

```
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

Set operations: ⋃ *pattern:A . B*

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

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.

```
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

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

fun_disjoint_Un:      [| f∈A->B; g∈C->D; A ∩ C = 0  |] ==>
                      (f ∪ g)∈(A ∪ C) -> (B ∪ D)

fun_disjoint_apply1: [| a∈A; f∈A->B; g∈C->D;  A∩C = 0 |] ==>
                      (f ∪ g)‘a = f‘a

fun_disjoint_apply2: [| c∈C; f∈A->B; g∈C->D;  A∩C = 0 |] ==>
                      (f ∪ 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 ZF theory provides many derived 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, $C =$ `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 proofs.

Figure 3.21 presents commutative, associative, distributive, and idempotency laws of union and intersection, along with other equations.

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

```
Int_absorb:         A ∩ A = A
Int_commute:        A ∩ B = B ∩ A
Int_assoc:          (A ∩ B) ∩ C  =  A ∩ (B ∩ C)
Int_Un_distrib:     (A ∪ B) ∩ C  =  (A ∩ C) ∪ (B ∩ C)

Un_absorb:          A ∪ A = A
Un_commute:         A ∪ B = B ∪ A
Un_assoc:           (A ∪ B) ∪ C  =  A ∪ (B ∪ C)
Un_Int_distrib:     (A ∩ B) ∪ C  =  (A ∪ C) ∩ (B ∪ C)

Diff_cancel:        A-A = 0
Diff_disjoint:      A ∩ (B-A) = 0
Diff_partition:     A ⊆ B ==> A ∪ (B-A) = B
double_complement:  [| A ⊆ B; B ⊆ C |] ==> (B - (C-A)) = A
Diff_Un:            A - (B ∪ C) = (A-B) ∩ (A-C)
Diff_Int:           A - (B ∩ C) = (A-B) ∪ (A-C)

Union_Un_distrib:   Union(A ∪ B) = Union(A) ∪ Union(B)
Inter_Un_distrib:   [| a ∈ A;  b ∈ B |] ==>
                    Inter(A ∪ B) = Inter(A) ∩ Inter(B)

Int_Union_RepFun:   A ∩ Union(B) = (⋃C ∈ B. A ∩ C)

Un_Inter_RepFun:    b ∈ B ==>
                    A ∪ Inter(B) = (⋂C ∈ B. A ∪ C)

SUM_Un_distrib1:    (SUM x ∈ A ∪ B. C(x)) =
                    (SUM x ∈ A. C(x)) ∪ (SUM x ∈ B. C(x))

SUM_Un_distrib2:    (SUM x ∈ C. A(x) ∪ B(x)) =
                    (SUM x ∈ C. A(x)) ∪ (SUM x ∈ C. B(x))

SUM_Int_distrib1:   (SUM x ∈ A ∩ B. C(x)) =
                    (SUM x ∈ A. C(x)) ∩ (SUM x ∈ B. C(x))

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

Figure 3.21: Equalities

```
bool_def:      bool == {0,1}
cond_def:      cond(b,c,d) == if b=1 then c else 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

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 ∪ {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)

InlI:      a ∈ A ==> Inl(a) ∈ A+B
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

sum_iff:  u ∈ A+B <-> (∃x∈A. u=Inl(x)) | (∃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. ∃a b. p=<a;b> & y=c(a,b)
qfsplit_def:    qfsplit(R,z) == ∃x y. z=<x;y> & R(x,y)
qconverse_def:  qconverse(r) == {z. w ∈ r, ∃x y. w=<x;y> & z=<y;x>}
QSigma_def:     QSigma(A,B)  == ⋃x ∈ A. ⋃y ∈ B(x). {<x;y>}

qsum_def:       A <+> B      == ({0} <*> A) ∪ ({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

3.6.1 Disjoint unions

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 [18].

3.6.2 Non-standard ordered pairs

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 [20].

3.6.3 Least and greatest fixedpoints

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 [17]. See Davey and Priestley [5] for more on the Knaster-Tarski Theorem and my paper [18] 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.

```
bnd_mono_def:  bnd_mono(D,h) ==
               h(D)⊆D & (∀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 ∪ 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

```
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 ∪ c ∈ Fin(A)
Fin_UnionI:     C ∈ Fin(Fin(A)) ==> Union(C) ∈ Fin(A)
Fin_subset:     [| c ⊆ b;  b ∈ Fin(A) |] ==> c ∈ Fin(A)
```

Figure 3.26: The finite set operator

These are useful for applying the Knaster-Tarski Fixedpoint Theorem. The proofs themselves are trivial applications of Isabelle's classical reasoner.

3.6.4 Finite sets and lists

Theory `Finite` (Figure 3.26) defines the finite set operator; *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 3.27 presents the set of lists over A, *list*(A). The definition employs Isabelle's datatype package, which defines the introduction and induction rules automatically, as well as the constructors, case operator (*list_case*) and recursion operator. The theory then defines the usual list functions by primitive recursion. See theory `List`.

3.6.5 Miscellaneous

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 3.28 displays many of their properties that have been proved. These results are fundamental to a treatment of equipollence and cardinality.

Theory *Univ* defines a 'universe' *univ*(A), which is used by the datatype package. This set contains A and the natural numbers. Vitally, it is closed under finite products: *univ*$(A) \times$ *univ*$(A) \subseteq$ *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.

symbol	*meta-type*	*priority*	*description*
`list`	$i \Rightarrow i$		lists over some set
`list_case`	$[i, [i,i] \Rightarrow i, i] \Rightarrow i$		conditional for $list(A)$
`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

```
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)

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 3.27: Lists

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) .
                        ∃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. ∀w∈A. ∀x∈A. f`w=f`x --> w=x }
surj_def: surj(A,B) == { f∈A->B . ∀y∈B. ∃x∈A. f`x=y }
bij_def:  bij(A,B)  == inj(A,B) ∩ 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 ∩ C = 0;  B ∩ D = 0 |] ==>
    (f ∪ g)∈bij(A ∪ C, B ∪ D)

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

Figure 3.28: Permutations

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

Figure 3.29: Some rewrite rules for set theory

Theory *QUniv* defines a 'universe' $\mathtt{quniv}(A)$, which is used by the datatype package to construct codatatypes such as streams. It is analogous to $\mathtt{univ}(A)$ (and is defined in terms of it) but is closed under the non-standard product and sum.

3.7 Automatic Tools

ZF provides the simplifier and the classical reasoner. Moreover it supplies a specialized tool to infer 'types' of terms.

3.7.1 Simplification and Classical Reasoning

ZF inherits simplification from FOL but adopts it for set theory. The extraction of rewrite rules takes the ZF 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 simpset used by *simp* contains congruence rules for all of ZF's binding operators. It contains all the conversion rules, such as *fst* and *snd*, as well as the rewrites shown in Fig. 3.29.

Classical reasoner methods such as *blast* and *auto* refer to a rich collection of built-in axioms for all the set-theoretic primitives.

3.7.2 Type-Checking Tactics

Isabelle/ZF provides simple tactics to help automate those proofs that are essentially type-checking. Such proofs are built by applying rules such as these:

```
[| ?P ==> ?a ∈ ?A; ~?P ==> ?b ∈ ?A |]
==> (if ?P then ?a else ?b) ∈ ?A

[| ?m ∈ nat; ?n ∈ nat |] ==> ?m #+ ?n ∈ nat

?a ∈ ?A ==> Inl(?a) ∈ ?A + ?B
```

In typical applications, the goal has the form $t \in ?A$: in other words, we have a specific term t and need to infer its 'type' by instantiating the set variable $?A$. Neither the simplifier nor the classical reasoner does this job well. The if-then-else rule, and many similar ones, can make the classical reasoner loop. The simplifier refuses (on principle) to instantiate variables during rewriting, so goals such as `i#+j ∈ ?A` are left unsolved.

The simplifier calls the type-checker to solve rewritten subgoals: this stage can indeed instantiate variables. If you have defined new constants and proved type-checking rules for them, then declare the rules using the attribute `TC` and the rest should be automatic. In particular, the simplifier will use type-checking to help satisfy conditional rewrite rules. Call the method `typecheck` to break down all subgoals using type-checking rules. You can add new type-checking rules temporarily like this:

apply *(typecheck add: inj_is_fun)*

3.8 Natural number and integer arithmetic

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` develops arithmetic on the natural numbers (Fig. 3.30). Addition, multiplication and subtraction are defined by primitive recursion. Division and remainder are defined by repeated subtraction, which requires well-founded recursion; the termination argument relies on the divisor's being non-zero. Many properties are proved: commutative, associative and distributive laws, identity and cancellation laws, etc. The most interesting result is perhaps the theorem $a \bmod b + (a/b) \times b = a$.

To minimize the need for tedious proofs of $t \in$ `nat`, the arithmetic operators coerce their arguments to be natural numbers. The function `natify` is defined such that `natify`$(n) = n$ if n is a natural number, `natify`$(succ(x)) =$ `succ`$($`natify`$(x))$ for all x, and finally `natify`$(x) = 0$ in all other cases. The benefit is that the addition, subtraction, multiplication, division and remainder operators always return natural numbers, regardless of their arguments. Algebraic laws (commutative, associative, distributive) are unconditional. Occurrences of `natify` as operands of those operators are simplified away. Any remaining occurrences can either be tolerated or else eliminated by proving that the argument is a natural number.

symbol	*meta-type*	*priority*	*description*
`nat`	i		set of natural numbers
`nat_case`	$[i, i \Rightarrow i, i] \Rightarrow i$		conditional 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} ∪ {succ(x). x ∈ r}

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

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)

add_0_natify:     0 #+ n = natify(n)
add_succ:         succ(m) #+ n = succ(m #+ n)

mult_type:        m #* n ∈ nat
mult_0:           0 #* n = 0
mult_succ:        succ(m) #* n = n #+ (m #* n)
mult_commute:     m #* n = n #* m
add_mult_dist:    (m #+ n) #* k = (m #* k) #+ (n #* k)
mult_assoc:       (m #* n) #* k = m #* (n #* k)
mod_div_equality: m ∈ nat ==> (m div n)#*n #+ m mod n = m
```

Figure 3.30: The natural numbers

symbol	*meta-type*	*priority*	*description*
`int`	i		set of integers
`$*`	$[i,i] \Rightarrow i$	Left 70	multiplication
`$+`	$[i,i] \Rightarrow i$	Left 65	addition
`$-`	$[i,i] \Rightarrow i$	Left 65	subtraction
`$<`	$[i,i] \Rightarrow o$	Left 50	$<$ on integers
`$<=`	$[i,i] \Rightarrow o$	Left 50	$\leq$ on integers

```
zadd_0_intify:    0 $+ n = intify(n)

zmult_type:       m $* n ∈ int
zmult_0:          0 $* n = 0
zmult_commute:    m $* n = n $* m
zadd_zmult_dist:   (m $+ n) $* k = (m $* k) $+ (n $* k)
zmult_assoc:      (m $* n) $* k = m $* (n $* k)
```

Figure 3.31: The integers

The simplifier automatically cancels common terms on the opposite sides of subtraction and of relations ($=$, $<$ and $\leq$). Here is an example:

```
1.  i #+ j #+ k #- j < k #+ l
apply simp
1.  natify(i) < natify(l)
```

Given the assumptions `i ∈nat` and `l ∈nat`, both occurrences of `natify` would be simplified away.

Theory `Int` defines the integers, as equivalence classes of natural numbers. Figure 3.31 presents a tidy collection of laws. In fact, a large library of facts is proved, including monotonicity laws for addition and multiplication, covering both positive and negative operands.

As with the natural numbers, the need for typing proofs is minimized. All the operators defined in Fig. 3.31 coerce their operands to integers by applying the function `intify`. This function is the identity on integers and maps other operands to zero.

Decimal notation is provided for the integers. Numbers, written as `#`*nnn* or `#-`*nnn*, are represented internally in two's-complement binary. Expressions involving addition, subtraction and multiplication of numeral constants are evaluated (with acceptable efficiency) by simplification. The simplifier also collects similar terms, multiplying them by a numerical coefficient. It also cancels occurrences of the same terms on the other side of the relational operators. Example:

```
1.  y $+ z $+ #-3 $* x $+ y $<= x $* #2 $+ z
apply simp
1.  #2 $* y $<= #5 $* x
```

For more information on the integers, please see the theories on directory `ZF/Integ`.

3.9 Datatype definitions

The `datatype` definition package of ZF constructs inductive datatypes similar to ML's. It can also construct coinductive datatypes (codatatypes), which are non-well-founded structures such as streams. It defines the set using a fixed-point construction and proves induction rules, as well as theorems for recursion and case combinators. It supplies mechanisms for reasoning about freeness. The datatype package can handle both mutual and indirect recursion.

3.9.1 Basics

A *`datatype`* definition has the following form:

$$\begin{array}{lrcl} \texttt{datatype} & t_1(A_1,\ldots,A_h) & = & \mathit{constructor}_1^1 \mid \ldots \mid \mathit{constructor}_{k_1}^1 \\ & & \vdots & \\ \texttt{and} & t_n(A_1,\ldots,A_h) & = & \mathit{constructor}_1^n \mid \ldots \mid \mathit{constructor}_{k_n}^n \end{array}$$

Here t_1, ..., t_n are identifiers and A_1, ..., A_h are variables: the datatype's parameters. Each constructor specification has the form

$$C \texttt{ ("}x_1\texttt{:}T_1\texttt{", ..., "}x_m\texttt{:}T_m\texttt{")}$$

Here C is the constructor name, and variables x_1, ..., x_m are the constructor arguments, belonging to the sets T_1, ..., T_m, respectively. Typically each T_j is either a constant set, a datatype parameter (one of A_1, ..., A_h) or a recursive occurrence of one of the datatypes, say $t_i(A_1,\ldots,A_h)$. More complex possibilities exist, but they are much harder to realize. Often, additional information must be supplied in the form of theorems.

A datatype can occur recursively as the argument of some function F. This is called a *nested* (or *indirect*) occurrence. It is only allowed if the datatype package is given a theorem asserting that F is monotonic. If the datatype has indirect occurrences, then Isabelle/ZF does not support recursive function definitions.

A simple example of a datatype is *`list`*, which is built-in, and is defined by

```
consts     list :: "i=>i"
datatype  "list(A)" = Nil | Cons ("a ∈ A", "l ∈ list(A)")
```

Note that the datatype operator must be declared as a constant first. However, the package declares the constructors. Here, *`Nil`* gets type i and *`Cons`* gets type $[i, i] \Rightarrow i$.

Trees and forests can be modelled by the mutually recursive datatype definition

```
consts
  tree :: "i=>i"
  forest :: "i=>i"
  tree_forest :: "i=>i"
datatype  "tree(A)"   = Tcons ("a∈A",  "f∈forest(A)")
and "forest(A)" = Fnil | Fcons ("t∈tree(A)",  "f∈forest(A)")
```

Here *tree*(A) is the set of trees over A, *forest*(A) is the set of forests over A, and *tree_forest*(A) is the union of the previous two sets. All three operators must be declared first.

The datatype *term*, which is defined by

```
consts     term :: "i=>i"
datatype  "term(A)" = Apply ("a ∈ A", "l ∈ list(term(A))")
  monos list_mono
  type_elims list_univ [THEN subsetD, elim_format]
```

is an example of nested recursion. (The theorem *list_mono* is proved in theory *List*, and the *term* example is developed in theory *Induct/Term*.)

Freeness of the constructors

Constructors satisfy *freeness* properties. Constructions are distinct, for example $Nil \neq Cons(a, l)$, and they are injective, for example $Cons(a, l) = Cons(a', l') \leftrightarrow a = a' \wedge l = l'$. Because the number of freeness is quadratic in the number of constructors, the datatype package does not prove them. Instead, it ensures that simplification will prove them dynamically: when the simplifier encounters a formula asserting the equality of two datatype constructors, it performs freeness reasoning.

Freeness reasoning can also be done using the classical reasoner, but it is more complicated. You have to add some safe elimination rules rules to the claset. For the *list* datatype, they are called *list.free_elims*. Occasionally this exposes the underlying representation of some constructor, which can be rectified using the command *unfold list.con_defs [symmetric]*.

Structural induction

The datatype package also provides structural induction rules. For datatypes without mutual or nested recursion, the rule has the form exemplified by *list.induct* in Fig. 3.27. For mutually recursive datatypes, the induction rule is supplied in two forms. Consider datatype *TF*. The rule *tree_forest.induct* performs induction over a single predicate *P*, which is presumed to be defined for both trees and forests:

```
[| x ∈ tree_forest(A);
   !!a f. [| a ∈ A; f ∈ forest(A); P(f) |] ==> P(Tcons(a, f));
   P(Fnil);
   !!f t. [| t ∈ tree(A); P(t); f ∈ forest(A); P(f) |]
          ==> P(Fcons(t, f))
|] ==> P(x)
```

The rule *tree_forest.mutual_induct* performs induction over two distinct predicates, *P_tree* and *P_forest*.

```
[| !!a f.
      [| a∈A; f∈forest(A); P_forest(f) |] ==> P_tree(Tcons(a,f));
   P_forest(Fnil);
   !!f t. [| t∈tree(A); P_tree(t); f∈forest(A); P_forest(f) |]
          ==> P_forest(Fcons(t, f))
|] ==> (∀za. za ∈ tree(A) --> P_tree(za)) &
    (∀za. za ∈ forest(A) --> P_forest(za))
```

For datatypes with nested recursion, such as the *term* example from above, things are a bit more complicated. The rule *term.induct* refers to the monotonic operator, *list*:

```
[| x ∈ term(A);
   !!a l. [| a∈A; l∈list(Collect(term(A), P)) |] ==> P(Apply(a,l))
|] ==> P(x)
```

The theory *Induct/Term.thy* derives two higher-level induction rules, one of which is particularly useful for proving equations:

```
[| t ∈ term(A);
   !!x zs. [| x ∈ A; zs ∈ list(term(A)); map(f, zs) = map(g, zs) |]
           ==> f(Apply(x, zs)) = g(Apply(x, zs))
|] ==> f(t) = g(t)
```

How this can be generalized to other nested datatypes is a matter for future research.

The *case* operator

The package defines an operator for performing case analysis over the datatype. For *list*, it is called *list_case* and satisfies the equations

```
list_case(f_Nil, f_Cons, []) = f_Nil
list_case(f_Nil, f_Cons, Cons(a, l)) = f_Cons(a, l)
```

Here *f_Nil* is the value to return if the argument is *Nil* and *f_Cons* is a function that computes the value to return if the argument has the form *Cons*(a, l). The function can be expressed as an abstraction, over patterns if desired (Sect. 3.5.4).

For mutually recursive datatypes, there is a single *case* operator. In the tree/forest example, the constant *tree_forest_case* handles all of the constructors of the two datatypes.

3.9.2 Defining datatypes

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

Codatatypes are declared like datatypes and are identical to them in every respect except that they have a coinduction rule instead of an induction rule. Note that while an induction rule has the effect of limiting the values contained in the set, a coinduction rule gives a way of constructing new values of the set.

Most of the theorems about datatypes become part of the default simpset. You never need to see them again because the simplifier applies them automatically.

Specialized methods for datatypes

Induction and case-analysis can be invoked using these special-purpose methods:

`induct_tac` x applies structural induction on variable x to subgoal 1, provided the type of x is a datatype. The induction variable should not occur among other assumptions of the subgoal.

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

`case_tac` x performs a case analysis for the variable x.

Both tactics can only be applied to a variable, whose typing must be given in some assumption, for example the assumption `x ∈ list(A)`. The tactics also work for the natural numbers (`nat`) and disjoint sums, although these sets were not defined using the datatype package. (Disjoint sums are not recursive, so only `case_tac` is available.)

Structured Isar methods are also available. Below, t stands for the name of the datatype.

`induct set:` t is the Isar induction tactic.

`cases set:` t is the Isar case-analysis tactic.

The theorems proved by a datatype declaration

Here are some more details for the technically minded. Processing the datatype declaration of a set t produces a name space t containing the following theorems:

`intros`	*the introduction rules*
`cases`	*the case analysis rule*
`induct`	*the standard induction rule*
`mutual_induct`	*the mutual induction rule, if needed*
`case_eqns`	*equations for the case operator*
`recursor_eqns`	*equations for the recursor*
`simps`	*the union of* `case_eqns` *and* `recursor_eqns`
`con_defs`	*definitions of the case operator and constructors*
`free_iffs`	*logical equivalences for proving freeness*
`free_elims`	*elimination rules for proving freeness*
`defs`	*datatype definition(s)*

Furthermore there is the theorem C for every constructor C; for example, the `list` datatype's introduction rules are bound to the identifiers `Nil` and `Cons`.

For a codatatype, the component `coinduct` is the coinduction rule, replacing the `induct` component.

See the theories `Induct/Ntree` and `Induct/Brouwer` for examples of infinitely branching datatypes. See theory `Induct/LList` for an example of a codatatype. Some of these theories illustrate the use of additional, undocumented features of the datatype package. Datatype definitions are reduced to inductive definitions, and the advanced features should be understood in that light.

3.9.3 Examples

The datatype of binary trees

Let us define the set `bt`(A) of binary trees over A. The theory must contain these lines:

```
consts   bt :: "i=>i"
datatype "bt(A)" = Lf | Br ("a∈A", "t1∈bt(A)", "t2∈bt(A)")
```

After loading the theory, we can prove some theorem. We begin by declaring the constructor's typechecking rules as simplification rules:

```
declare bt.intros [simp]
```

Our first example is the theorem that no tree equals its left branch. To make the inductive hypothesis strong enough, the proof requires a quantified induction formula, but the `rule_format` attribute will remove the quantifiers before the theorem is stored.

```
lemma Br_neq_left [rule_format]: "l∈bt(A) ==> ∀x r. Br(x,l,r)≠l"
 1. l ∈ bt(A) ⟹ ∀x r. Br(x, l, r) ≠ l
```

This can be proved by the structural induction tactic:

```
  apply (induct_tac l)
1. ∀x r. Br(x, Lf, r) ≠ Lf
2. ⋀a t1 t2.
      ⟦a ∈ A; t1 ∈ bt(A); ∀x r. Br(x, t1, r) ≠ t1; t2 ∈ bt(A);
         ∀x r. Br(x, t2, r) ≠ t2⟧
      ⟹ ∀x r. Br(x, Br(a, t1, t2), r) ≠ Br(a, t1, t2)
```

Both subgoals are proved using `auto`, which performs the necessary freeness reasoning.

```
  apply auto
No subgoals!
done
```

An alternative proof uses Isar's fancy `induct` method, which automatically quantifies over all free variables:

```
lemma Br_neq_left': "l ∈ bt(A) ==> (!!x r. Br(x, l, r) ≠ l)"
  apply (induct set: bt)
1. ⋀x r. Br(x, Lf, r) ≠ Lf
2. ⋀a t1 t2 x r.
      ⟦a ∈ A; t1 ∈ bt(A); ⋀x r. Br(x, t1, r) ≠ t1; t2 ∈ bt(A);
         ⋀x r. Br(x, t2, r) ≠ t2⟧
      ⟹ Br(x, Br(a, t1, t2), r) ≠ Br(a, t1, t2)
```

Compare the form of the induction hypotheses with the corresponding ones in the previous proof. As before, to conclude requires only `auto`.

When there are only a few constructors, we might prefer to prove the freenness theorems for each constructor. This is simple:

```
lemma Br_iff: "Br(a,l,r) = Br(a',l',r') <-> a=a' & l=l' & r=r'"
  by (blast elim!: bt.free_elims)
```

Here we see a demonstration of freeness reasoning using `bt.free_elims`, but simpler still is just to apply `auto`.

An `inductive_cases` declaration generates instances of the case analysis rule that have been simplified using freeness reasoning.

```
inductive_cases Br_in_bt: "Br(a, l, r) ∈ bt(A)"
```

The theorem just created is

```
⟦Br(a, l, r) ∈ bt(A); ⟦a ∈ A; l ∈ bt(A); r ∈ bt(A)⟧ ⟹ Q⟧ ⟹ Q.
```

It is an elimination rule that from $\texttt{Br}(a, l, r) \in \texttt{bt}(A)$ lets us infer $a \in A$, $l \in \texttt{bt}(A)$ and $r \in \texttt{bt}(A)$.

Mixfix syntax in datatypes

Mixfix syntax is sometimes convenient. The theory *Induct/PropLog* makes a deep embedding of propositional logic:

```
consts     prop :: i
datatype  "prop" = Fls
                 | Var ("n ∈ nat")                ("#_" [100] 100)
                 | "=>" ("p ∈ prop", "q ∈ prop")   (infixr 90)
```

The second constructor has a special $\#n$ syntax, while the third constructor is an infixed arrow.

A giant enumeration type

This example shows a datatype that consists of 60 constructors:

```
consts  enum :: i
datatype
  "enum" = C00 | C01 | C02 | C03 | C04 | C05 | C06 | C07 | C08 | C09
         | C10 | C11 | C12 | C13 | C14 | C15 | C16 | C17 | C18 | C19
         | C20 | C21 | C22 | C23 | C24 | C25 | C26 | C27 | C28 | C29
         | C30 | C31 | C32 | C33 | C34 | C35 | C36 | C37 | C38 | C39
         | C40 | C41 | C42 | C43 | C44 | C45 | C46 | C47 | C48 | C49
         | C50 | C51 | C52 | C53 | C54 | C55 | C56 | C57 | C58 | C59
end
```

The datatype package scales well. Even though all properties are proved rather than assumed, full processing of this definition takes around two seconds (on a 1.8GHz machine). The constructors have a balanced representation, related to binary notation, so freeness properties can be proved fast.

```
lemma "C00 ≠ C01"
  by simp
```

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

3.9.4 Recursive function definitions

Datatypes come with a uniform way of defining functions, **primitive recursion**. Such definitions rely on the recursion operator defined by the datatype package. Isabelle proves the desired recursion equations as theorems.

In principle, one could introduce primitive recursive functions by asserting their reduction rules as axioms. Here is a dangerous way of defining a recursive function over binary trees:

```
consts  n_nodes :: "i => i"
axioms
  n_nodes_Lf: "n_nodes(Lf) = 0"
  n_nodes_Br: "n_nodes(Br(a,l,r)) = succ(n_nodes(l) #+ n_nodes(r))"
```

Asserting axioms brings the danger of accidentally introducing contradictions. It should be avoided whenever possible.

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

```
consts  n_nodes :: "i => i"
primrec
  "n_nodes(Lf) = 0"
  "n_nodes(Br(a, l, r)) = succ(n_nodes(l) #+ n_nodes(r))"
```

Isabelle will now derive the two equations from a low-level definition based upon well-founded recursion. If they do not define a legitimate recursion, then Isabelle will reject the declaration.

Syntax of recursive definitions

The general form of a primitive recursive definition is

primrec
reduction rules

where *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 at most one reduction rule for each constructor. The order is immaterial. For missing constructors, the function is defined to return zero.

All reduction rules are added to the default simpset. If you would like to refer to some rule by name, then you must prefix the rule with an identifier. These identifiers, like those in the *rules* section of a theory, will be visible in proof scripts.

The reduction rules become part of the default simpset, which leads to short proof scripts:

```
lemma n_nodes_type [simp]: "t ∈ bt(A) ==> n_nodes(t) ∈ nat"
  by (induct_tac t, auto)
```

You can even use the *primrec* form with non-recursive datatypes and with codatatypes. Recursion is not allowed, but it provides a convenient syntax for defining functions by cases.

Example: varying arguments

All arguments, other than the recursive one, must be the same in each equation and in each recursive call. To get around this restriction, use explict λ-abstraction and function application. For example, let us define the tail-recursive version of `n_nodes`, using an accumulating argument for the counter. The second argument, k, varies in recursive calls.

```
consts  n_nodes_aux :: "i => i"
primrec
  "n_nodes_aux(Lf) = (λk ∈ nat. k)"
  "n_nodes_aux(Br(a,l,r)) =
      (λk ∈ nat. n_nodes_aux(r) ` (n_nodes_aux(l) ` succ(k)))"
```

Now `n_nodes_aux(t) ` k` is our function in two arguments. We can prove a theorem relating it to `n_nodes`. Note the quantification over `k ∈ nat`:

```
lemma n_nodes_aux_eq [rule_format]:
     "t ∈ bt(A) ==> ∀k ∈ nat. n_nodes_aux(t)`k = n_nodes(t) #+ k"
  by (induct_tac t, simp_all)
```

Now, we can use `n_nodes_aux` to define a tail-recursive version of `n_nodes`:

```
constdefs
  n_nodes_tail :: "i => i"
   "n_nodes_tail(t) == n_nodes_aux(t) ` 0"
```

It is easy to prove that `n_nodes_tail` is equivalent to `n_nodes`:

```
lemma "t ∈ bt(A) ==> n_nodes_tail(t) = n_nodes(t)"
 by (simp add: n_nodes_tail_def n_nodes_aux_eq)
```

3.10 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 formalise 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. It behaves identially to the analogous inductive definition except that instead of an induction rule there is a coinduction rule. Its

treatment of coinduction is described in detail in a separate paper,[2] which you might refer to for background information.

3.10.1 The syntax of a (co)inductive definition

An inductive definition has the form

`inductive`	
`domains`	*domain declarations*
`intros`	*introduction rules*
`monos`	*monotonicity theorems*
`con_defs`	*constructor definitions*
`type_intros`	*introduction rules for type-checking*
`type_elims`	*elimination rules for type-checking*

A coinductive definition is identical, but starts with the keyword `coinductive`.

The `monos`, `con_defs`, `type_intros` and `type_elims` sections are optional. If present, each is specified as a list of theorems, which may contain Isar attributes as usual.

domain declarations are items of the form $string \subseteq string$, associating each recursive set with its domain. (The domain is some existing set that is large enough to hold the new set being defined.)

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.

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!

constructor definitions contain definitions of constants appearing in the introduction rules. The (co)datatype package supplies the constructors' definitions here. Most (co)inductive definitions omit this section; one exception is the primitive recursive functions example; see theory `Induct/Primrec`.

type_intros consists of introduction rules for type-checking the definition: for demonstrating that the new set is included in its domain. (The proof uses depth-first search.)

[2] It appeared in CADE [17]; a longer version is distributed with Isabelle as *A Fixedpoint Approach to (Co)Inductive and (Co)Datatype Definitions*.

type_elims consists of elimination rules for type-checking the definition. They are presumed to be safe and are applied as often as possible prior to the `type_intros` search.

The package has a few restrictions:

- The theory must separately declare the recursive sets as constants.
- The names of the recursive sets must be identifiers, not infix operators.
- 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`.

3.10.2 Example of an inductive definition

Below, we shall see how Isabelle/ZF defines the finite powerset operator. The first step is to declare the constant `Fin`. Then we must declare it inductively, with two introduction rules:

```
consts  Fin :: "i=>i"
inductive
  domains   "Fin(A)" ⊆ "Pow(A)"
  intros
    emptyI:  "0 ∈ Fin(A)"
    consI:   "[| a ∈ A;  b ∈ Fin(A) |] ==> cons(a,b) ∈ Fin(A)"
  type_intros  empty_subsetI cons_subsetI PowI
  type_elims   PowD [THEN revcut_rl]
```

The resulting theory contains a name space, called `Fin`. The `Fin` A introduction rules can be referred to collectively as `Fin.intros`, and also individually as `Fin.emptyI` and `Fin.consI`. The induction rule is `Fin.induct`.

The chief problem with making (co)inductive definitions involves type-checking the rules. Sometimes, additional theorems need to be supplied under `type_intros` or `type_elims`. If the package fails when trying to prove your introduction rules, then set the flag `trace_induct` to `true` and try again. (See the manual *A Fixedpoint Approach ...* for more discussion of type-checking.)

In the example above, `Pow`(A) is given as the domain of `Fin`(A), for obviously every finite subset of A is a subset of A. However, the inductive definition package can only prove that given a few hints. Here is the output that results (with the flag set) when the `type_intros` and `type_elims` are omitted from the inductive definition above:

```
Inductive definition Finite.Fin
Fin(A) ==
```

```
lfp(Pow(A),
    %X. z∈Pow(A) . z = 0 | (∃a b. z = cons(a,b) & a∈A & b∈X))
  Proving monotonicity...
  Proving the introduction rules...
The type-checking subgoal:
0 ∈ Fin(A)
 1. 0 ∈ Pow(A)
The subgoal after monos, type_elims:
0 ∈ Fin(A)
 1. 0 ∈ Pow(A)
*** prove_goal: tactic failed
```

We see the need to supply theorems to let the package prove $\emptyset \in \texttt{Pow}(A)$. Restoring the *type_intros* but not the *type_elims*, we again get an error message:

```
The type-checking subgoal:
0 ∈ Fin(A)
 1. 0 ∈ Pow(A)
The subgoal after monos, type_elims:
0 ∈ Fin(A)
 1. 0 ∈ Pow(A)
The type-checking subgoal:
cons(a, b) ∈ Fin(A)
 1. [| a ∈ A; b ∈ Fin(A) |] ==> cons(a, b) ∈ Pow(A)
The subgoal after monos, type_elims:
cons(a, b) ∈ Fin(A)
 1. [| a ∈ A; b ∈ Pow(A) |] ==> cons(a, b) ∈ Pow(A)
*** prove_goal: tactic failed
```

The first rule has been type-checked, but the second one has failed. The simplest solution to such problems is to prove the failed subgoal separately and to supply it under *type_intros*. The solution actually used is to supply, under *type_elims*, a rule that changes $b \in \texttt{Pow}(A)$ to $b \subseteq A$; together with *cons_subsetI* and *PowI*, it is enough to complete the type-checking.

3.10.3 Further examples

An inductive definition may involve arbitrary monotonic operators. Here is a standard example: the accessible part of a relation. Note the use of *Pow* in the introduction rule and the corresponding mention of the rule *Pow_mono* in the *monos* list. If the desired rule has a universally quantified premise, usually the effect can be obtained using *Pow*.

```
consts  acc :: "i => i"
inductive
  domains "acc(r)" ⊆ "field(r)"
  intros
    vimage:  "[| r-``{a} ∈ Pow(acc(r)); a ∈ field(r) |]
```

```
            ==> a ∈ acc(r)"
  monos  Pow_mono
```

Finally, here are some coinductive definitions. We begin by defining lazy (potentially infinite) lists as a codatatype:

```
consts  llist  :: "i=>i"
codatatype
  "llist(A)" = LNil | LCons ("a ∈ A", "l ∈ llist(A)")
```

The notion of equality on such lists is modelled as a bisimulation:

```
consts  lleq :: "i=>i"
coinductive
  domains "lleq(A)" <= "llist(A) * llist(A)"
  intros
    LNil:  "<LNil, LNil> ∈ lleq(A)"
    LCons: "[| a ∈ A; <l,l'> ∈ lleq(A) |]
            ==> <LCons(a,l), LCons(a,l')> ∈ lleq(A)"
  type_intros  llist.intros
```

This use of *type_intros* is typical: the relation concerns the codatatype *llist*, so naturally the introduction rules for that codatatype will be required for type-checking the rules.

The Isabelle distribution contains many other inductive definitions. Simple examples are collected on subdirectory *ZF/Induct*. The directory *Coind* and the theory *ZF/Induct/LList* contain coinductive definitions. Larger examples may be found on other subdirectories of *ZF*, such as *IMP*, and *Resid*.

3.10.4 Theorems generated

Each (co)inductive set defined in a theory file generates a name space containing the following elements:

intros	*the introduction rules*
elim	*the elimination (case analysis) rule*
induct	*the standard induction rule*
mutual_induct	*the mutual induction rule, if needed*
defs	*definitions of inductive sets*
bnd_mono	*monotonicity property*
dom_subset	*inclusion in 'bounding set'*

Furthermore, each introduction rule is available under its declared name. For a codatatype, the component *coinduct* is the coinduction rule, replacing the *induct* component.

Recall that the `inductive_cases` declaration generates simplified instances of the case analysis rule. It is as useful for inductive definitions as it is for datatypes. There are many examples in the theory *Induct/Comb*, which is discussed at length elsewhere [19]. The theory first defines the datatype *comb* of combinators:

```
consts comb :: i
datatype  "comb" = K
                 | S
                 | "#" ("p ∈ comb", "q ∈ comb")   (infixl 90)
```

The theory goes on to define contraction and parallel contraction inductively. Then the theory *Induct/Comb.thy* defines special cases of contraction, such as this one:

inductive_cases *K_contractE [elim!]: "K -1-> r"*

The theorem just created is *K -1-> r* $\Longrightarrow$ *Q*, which expresses that the combinator *K* cannot reduce to anything. (From the assumption *K-1->r*, we can conclude any desired formula *Q*.) Similar elimination rules for *S* and application are also generated. The attribute *elim!* shown above supplies the generated theorem to the classical reasoner. This mode of working allows effective reasoniung about operational semantics.

3.11 The outer reaches of set theory

The constructions of the natural numbers and lists use a suite of operators for handling recursive function definitions. I have described the developments in detail elsewhere [18]. 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 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 reflexivity, 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 [10, page 29]. None of these results assume the Axiom of Choice, which complicates their proofs considerably.

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.

3.12 The examples directories

Directory *HOL/IMP* contains a mechanised version of a semantic equivalence proof taken from Winskel [25]. 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 [16, 18, 21].

- File *misc.ML* contains miscellaneous examples such as Cantor's Theorem, the Schröder-Bernstein Theorem and the 'Composition of homomorphisms' challenge [3].
- Theory *Ramsey* proves the finite exponent 2 version of Ramsey's Theorem, following Basin and Kaufmann's presentation [2].
- 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 0.3 seconds.
- Theory *BT* defines the recursive data structure $\mathtt{bt}(A)$, 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 [18]. This illustrates datatype definitions, inductive definitions, structural induction and rule induction.
- Theory *ListN* inductively defines the lists of n elements [13].
- Theory *Acc* inductively defines the accessible part of a relation [13].
- 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 [4].
- Theory *LList* defines lazy lists and a coinduction principle for proving equations between them.

3.13 A proof about powersets

To demonstrate high-level reasoning about subsets, let us prove the equation $\mathtt{Pow}(A) \cap \mathtt{Pow}(B) = \mathtt{Pow}(A \cap B)$. 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:

```
lemma "Pow(A Int B) = Pow(A) Int Pow(B)"
 1. Pow(A ∩ B) = Pow(A) ∩ Pow(B)
apply (rule equalityI)
 1. Pow(A ∩ B) ⊆ Pow(A) ∩ Pow(B)
 2. Pow(A) ∩ Pow(B) ⊆ Pow(A ∩ B)
```

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

```
apply (rule Int_greatest)
 1. Pow(A ∩ B) ⊆ Pow(A)
 2. Pow(A ∩ B) ⊆ Pow(B)
 3. Pow(A) ∩ Pow(B) ⊆ Pow(A ∩ B)
```

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

```
apply (rule Int_lower1 [THEN Pow_mono])
 1. Pow(A ∩ B) ⊆ Pow(B)
 2. Pow(A) ∩ Pow(B) ⊆ Pow(A ∩ B)
apply (rule Int_lower2 [THEN Pow_mono])
 1. Pow(A) ∩ Pow(B) ⊆ Pow(A ∩ B)
```

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

```
apply (rule subsetI)
 1. ⋀x. x ∈ Pow(A) ∩ Pow(B) ⟹ x ∈ Pow(A ∩ B)
```

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

```
apply (erule IntE)
 1. ⋀x. ⟦x ∈ Pow(A); x ∈ Pow(B)⟧ ⟹ x ∈ Pow(A ∩ B)
```

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

apply *(rule PowI)*
 1. ⋀x. ⟦x ∈ Pow(A); x ∈ Pow(B)⟧ ⟹ x ⊆ A ∩ B

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

apply *(drule PowD)+*
 1. ⋀x. ⟦x ⊆ A; x ⊆ B⟧ ⟹ x ⊆ A ∩ B

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

apply *(rule Int_greatest)*
 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:

apply *(assumption+)*
done

We could have performed this proof instantly by calling `blast`:

lemma *"Pow(A Int B) = Pow(A) Int Pow(B)"*
by

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

3.14 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*:

lemma *"C⊆D ==> Union(C) ⊆ Union(D)"*
apply *(rule subsetI)*
 1. ⋀x. ⟦C ⊆ D; x ∈ ⋃C⟧ ⟹ x ∈ ⋃D

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

apply *(erule UnionE)*
 1. ⋀x B. ⟦C ⊆ D; x ∈ B; B ∈ C⟧ ⟹ x ∈ ⋃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.

apply *(rule UnionI)*
 1. ⋀x B. ⟦C ⊆ D; x ∈ B; B ∈ C⟧ ⟹ ?B2(x, B) ∈ D
 2. ⋀x B. ⟦C ⊆ D; x ∈ B; B ∈ C⟧ ⟹ x ∈ ?B2(x, B)

Combining the rule *subsetD* with the assumption $C \subseteq D$ yields $?a \in C \Longrightarrow ?a \in D$, which reduces subgoal 1. Note that *erule* removes the subset assumption.

```
apply (erule subsetD)
 1. ⋀x B. ⟦x ∈ B; B ∈ C⟧ ⟹ ?B2(x, B) ∈ C
 2. ⋀x B. ⟦C ⊆ D; x ∈ B; B ∈ C⟧ ⟹ x ∈ ?B2(x, B)
```

The rest is routine. Observe how the first call to *assumption* instantiates *?B2(x,B)* to *B*.

```
apply assumption
 1. ⋀x B. ⟦C ⊆ D; x ∈ B; B ∈ C⟧ ⟹ x ∈ B
apply assumption
No subgoals!
done
```

Again, *blast* can prove this theorem in one step.

The theory *ZF/equalities.thy* has many similar proofs. Reasoning about general intersection can be difficult because of its anomalous behaviour on the empty set. However, *blast* copes well with these. Here is a typical example, borrowed from Devlin [6, page 12]:

$$a \in C \Longrightarrow \bigcap_{x\in C}\Big(A(x) \cap B(x)\Big) = \Big(\bigcap_{x\in C} A(x)\Big) \cap \Big(\bigcap_{x\in C} B(x)\Big)$$

3.15 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 $(f \cup g)`a = f`a$:

```
lemma "[| a ∈ A;  f ∈ A->B;  g ∈ C->D;  A ∩ C = 0 |]
       ==> (f ∪ g)`a = f`a"
```

Using *apply_equality*, we reduce the equality to reasoning about ordered pairs. The second subgoal is to verify that *f ∪ g* is a function, since *Pi(?A,?B)* denotes a dependent function space.

```
apply (rule apply_equality)
 1. ⟦a ∈ A; f ∈ A → B; g ∈ C → D; A ∩ C = 0⟧
    ⟹ ⟨a, f ` a⟩ ∈ f ∪ g
 2. ⟦a ∈ A; f ∈ A → B; g ∈ C → D; A ∩ C = 0⟧
    ⟹ f ∪ g ∈ Pi(?A, ?B)
```

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

apply *(rule UnI1)*
1. ⟦a ∈ A; f ∈ A → B; g ∈ C → D; A ∩ C = 0⟧ ⟹ ⟨a, f ‘ a⟩ ∈ f
2. ⟦a ∈ A; f ∈ A → B; g ∈ C → D; A ∩ C = 0⟧
   ⟹ f ∪ g ∈ Pi(?A, ?B)

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

apply *(rule apply_Pair)*
1. ⟦a ∈ A; f ∈ A → B; g ∈ C → D; A ∩ C = 0⟧ ⟹ f ∈ Pi(?A2,?B2)
2. ⟦a ∈ A; f ∈ A → B; g ∈ C → D; A ∩ C = 0⟧ ⟹ a ∈ ?A2
3. ⟦a ∈ A; f ∈ A → B; g ∈ C → D; A ∩ C = 0⟧
   ⟹ f ∪ g ∈ Pi(?A, ?B)

Using the assumptions $f \in A \to B$ 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` gets instantiated to `A`.

apply *assumption*
1. ⟦a ∈ A; f ∈ A → B; g ∈ C → D; A ∩ C = 0⟧ ⟹ a ∈ A
2. ⟦a ∈ A; f ∈ A → B; g ∈ C → D; A ∩ C = 0⟧
   ⟹ f ∪ g ∈ Pi(?A, ?B)

apply *assumption*
1. ⟦a ∈ A; f ∈ A → B; g ∈ C → D; A ∩ C = 0⟧
   ⟹ f ∪ g ∈ Pi(?A, ?B)

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

apply *(rule fun_disjoint_Un)*
1. ⟦a ∈ A; f ∈ A → B; g ∈ C → D; A ∩ C = 0⟧ ⟹ f ∈ ?A3 → ?B3
2. ⟦a ∈ A; f ∈ A → B; g ∈ C → D; A ∩ C = 0⟧ ⟹ g ∈ ?C3 → ?D3
3. ⟦a ∈ A; f ∈ A → B; g ∈ C → D; A ∩ C = 0⟧ ⟹ ?A3 ∩ ?C3 = 0

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

apply *assumption*
1. ⟦a ∈ A; f ∈ A → B; g ∈ C → D; A ∩ C = 0⟧ ⟹ g ∈ ?C3 → ?D3
2. ⟦a ∈ A; f ∈ A → B; g ∈ C → D; A ∩ C = 0⟧ ⟹ A ∩ ?C3 = 0

apply *assumption*
1. ⟦a ∈ A; f ∈ A → B; g ∈ C → D; A ∩ C = 0⟧ ⟹ A ∩ C = 0

apply *assumption*
No subgoals!
done

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

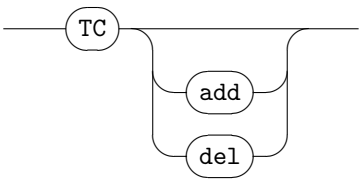
Chapter 4

Some Isar language elements

4.1 Type checking

The ZF logic is essentially untyped, so the concept of "type checking" is performed as logical reasoning about set-membership statements. A special method assists users in this task; a version of this is already declared as a "solver" in the standard Simplifier setup.

print_tcset*	:	*context* $\to$
typecheck	:	*method*
TC	:	*attribute*



print_tcset prints the collection of typechecking rules of the current context.

typecheck attempts to solve any pending type-checking problems in subgoals.

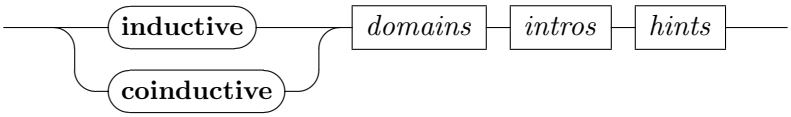
TC adds or deletes type-checking rules from the context.

4.2 (Co)Inductive sets and datatypes

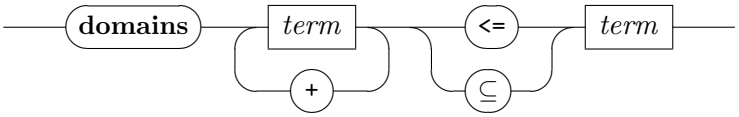
4.2.1 Set definitions

In ZF everything is a set. The generic inductive package also provides a specific view for "datatype" specifications. Coinductive definitions are available in both cases, too.

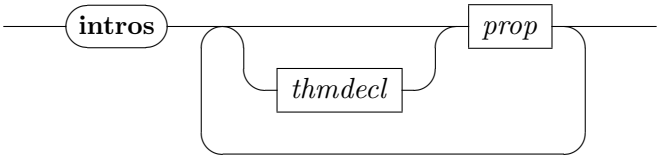
inductive : *theory* → *theory*
coinductive : *theory* → *theory*
datatype : *theory* → *theory*
codatatype : *theory* → *theory*



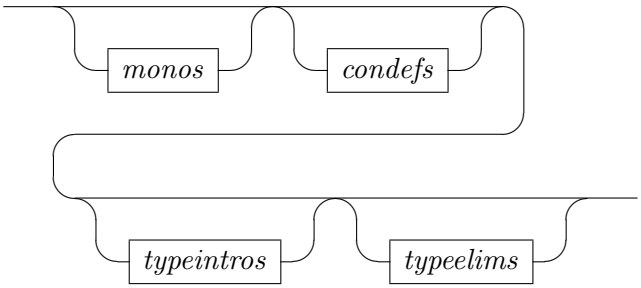
domains



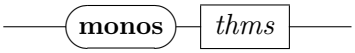
intros



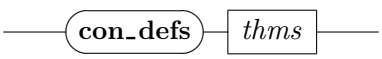
hints



monos



condefs

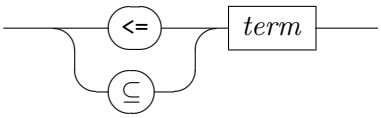


typeintros

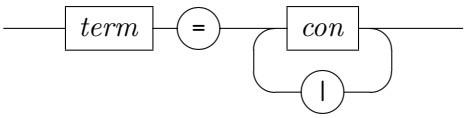
typeelims

In the following syntax specification *monos*, *typeintros*, and *typeelims* are the same as above.

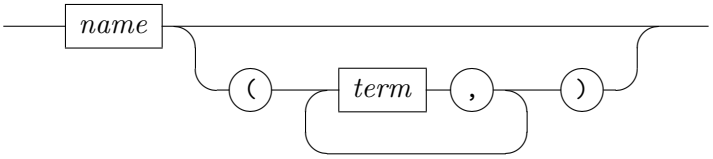
domain



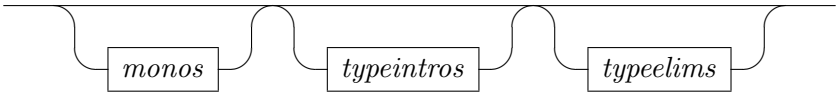
dtspec



con



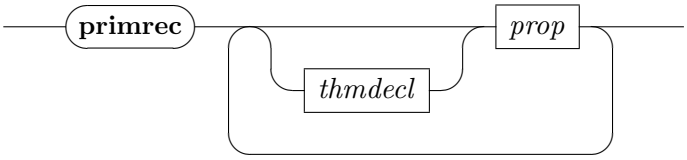
hints



See [14] for further information on inductive definitions in ZF, but note that this covers the old-style theory format.

4.2.2 Primitive recursive functions

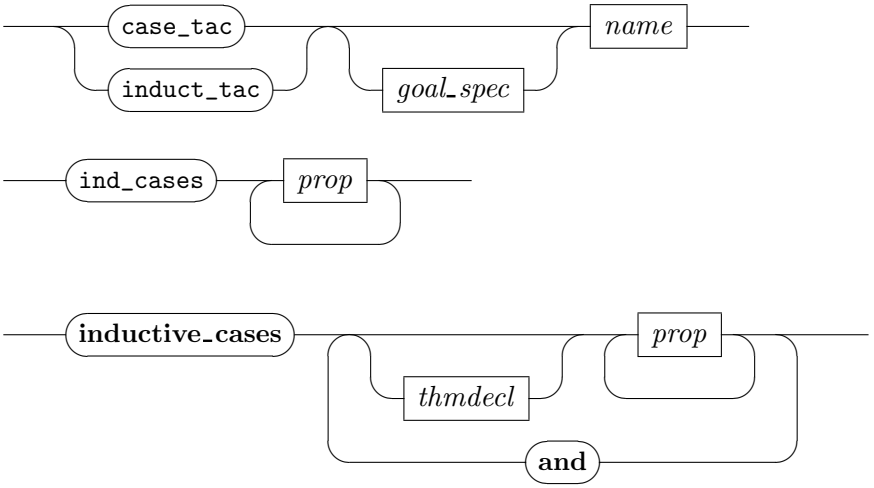
primrec : *theory* → *theory*



4.2.3 Cases and induction: emulating tactic scripts

The following important tactical tools of Isabelle/ZF have been ported to Isar. These should not be used in proper proof texts.

*case_tac**	:	*method*
*induct_tac**	:	*method*
*ind_cases**	:	*method*
inductive_cases	:	*theory* $\to$ *theory*



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] 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.

[3] 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.

[4] 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.

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

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

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

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

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

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

[11] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

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

[13] Christine Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. In M. Bezem and J.F. Groote, editors, *Typed Lambda Calculi and Applications*, LNCS 664, pages 328–345. Springer, 1993.

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

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

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

[17] 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.

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

[19] Lawrence C. Paulson. Generic automatic proof tools. In Robert Veroff, editor, *Automated Reasoning and its Applications: Essays in Honor of Larry Wos*, chapter 3. MIT Press, 1997.

[20] Lawrence C. Paulson. Final coalgebras as greatest fixed points in ZF set theory. *Mathematical Structures in Computer Science*, 9(5):545–567, 1999.

[21] Lawrence C. Paulson. A fixedpoint approach to (co)inductive and (co)datatype definitions. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction: Essays in Honor of Robin Milner*, pages 187–211. MIT Press, 2000.

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

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

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

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

Index