Isabelle's Logics

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

With Contributions by Tobias Nipkow and Markus Wenzel[1]

12 February 2013

[1]Markus Wenzel made numerous improvements. Sara Kalvala contributed Chap. 3. Philippe de Groote wrote the first version of the logic LK. Tobias Nipkow developed LCF and Cube. Martin Coen developed Modal with assistance from Rajeev Goré. The research has been funded by the EPSRC (grants GR/G53279, GR/H40570, GR/K57381, GR/K77051, GR/M75440) and by ESPRIT (projects 3245: Logical Frameworks, and 6453: Types), and by the DFG Schwerpunktprogramm *Deduktion*.

Contents

Preface

Several logics come with Isabelle. Many of them are sufficiently developed to serve as comfortable reasoning environments. They are also good starting points for defining new logics. Each logic is distributed with sample proofs, some of which are described in this document.

`HOL` is currently the best developed Isabelle object-logic, including an extensive library of (concrete) mathematics, and various packages for advanced definitional concepts (like (co-)inductive sets and types, well-founded recursion etc.). The distribution also includes some large applications. See the separate manual *Isabelle's Logics: HOL*. There is also a comprehensive tutorial on Isabelle/HOL available.

`ZF` provides another starting point for applications, with a slightly less developed library than `HOL`. `ZF`'s definitional packages are similar to those of `HOL`. Untyped `ZF` set theory provides more advanced constructions for sets than simply-typed `HOL`. `ZF` is built on `FOL` (first-order logic), both are described in a separate manual *Isabelle's Logics: FOL and ZF* [10].

There are some further logics distributed with Isabelle:

`CCL` is Martin Coen's Classical Computational Logic, which is the basis of a preliminary method for deriving programs from proofs [1]. It is built upon classical FOL.

`LCF` is a version of Scott's Logic for Computable Functions, which is also implemented by the LCF system [11]. It is built upon classical FOL.

`HOLCF` is a version of LCF, defined as an extension of `HOL`. See [8] for more details on `HOLCF`.

`CTT` is a version of Martin-Löf's Constructive Type Theory [9], with extensional equality. Universes are not included.

`Cube` is Barendregt's λ-cube.

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

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

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

`ILL` implements intuitionistic linear logic.

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

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

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

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

Chapter 1

Syntax definitions

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

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

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

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

A **binder** is a symbol associated with a constant of type $(\sigma \Rightarrow \tau) \Rightarrow \tau'$. For instance, we may declare $\forall$ as a binder for the constant All, which has type $(\alpha \Rightarrow o) \Rightarrow o$. This defines the syntax $\forall x\,.\,t$ to mean $All(\lambda x\,.\,t)$. We can also write $\forall x_1 \ldots x_m\,.\,t$ to abbreviate $\forall x_1\,.\,\ldots \forall x_m\,.\,t$; this is possible for any constant provided that τ and τ' are the same type. The Hilbert description operator $\varepsilon x\,.\,P\,x$ has type $(\alpha \Rightarrow bool) \Rightarrow \alpha$ and normally binds only one variable. ZF's bounded quantifier $\forall x \in A . P(x)$ cannot be declared as a binder because it has type $[i, i \Rightarrow o] \Rightarrow o$. The syntax for binders allows type constraints on bound variables, as in

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

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

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

Chapter 2

First-Order Sequent Calculus

The theory `LK` implements classical first-order logic through Gentzen's sequent calculus (see Gallier [4] or Takeuti [13]). Resembling the method of semantic tableaux, the calculus is well suited for backwards proof. Assertions have the form $\Gamma \vdash \Delta$, where Γ and Δ are lists of formulae. Associative unification, simulated by higher-order unification, handles lists (§2.7 presents details, if you are interested).

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

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

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

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

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

2.1 Syntax and rules of inference

Figure 2.1 gives the syntax for `LK`, which is complicated by the representation of sequents. Type $sobj \Rightarrow sobj$ represents a list of formulae.

The **definite description** operator $\iota x \,.\, P[x]$ stands for some a satisfying $P[a]$, if one exists and is unique. Since all terms in LK denote something, a description is always meaningful, but we do not know its value unless $P[x]$ defines it uniquely. The Isabelle notation is `THE` x`.` $P[x]$. The corresponding rule (Fig. 2.4) does not entail the Axiom of Choice because it requires uniqueness.

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

Constants

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

Binders

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

Infixes

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

Translations

Figure 2.1: Syntax of `LK`

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

Figure 2.2: Grammar of LK

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

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

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

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

Structural rules

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

Equality rules

Figure 2.3: Basic Rules of `LK`

Conditional expressions are available with the notation

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

Figure 2.2 presents the grammar of LK. Traditionally, Γ and Δ are meta-variables for sequences. In Isabelle's notation, the prefix `$` on a term makes it range over sequences. In a sequent, anything not prefixed by `$` is taken as a formula.

The notation `<<`*sequence*`>>` stands for a sequence of formulæ. For example, you can declare the constant `imps` to consist of two implications:

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

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

```
True_def    True  == False-->False
iff_def     P<->Q == (P-->Q) & (Q-->P)

conjR   [| $H|- $E, P, $F;  $H|- $E, Q, $F |] ==> $H|- $E, P&Q, $F
conjL   $H, P, Q, $G |- $E ==> $H, P & Q, $G |- $E

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

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

notR    $H, P |- $E, $F ==> $H |- $E, ~P, $F
notL    $H, $G |- $E, P ==> $H, ~P, $G |- $E

FalseL  $H, False, $G |- $E

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

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

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

Logical rules

Figure 2.4: Rules of LK

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

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

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

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

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

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

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

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

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

Figure 2.5: Derived rules for LK

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

Figures 2.3 and 2.4 present the rules of theory LK. The connective $\leftrightarrow$ is defined using $\land$ and $\rightarrow$. The axiom for basic sequents is expressed in a form that provides automatic thinning: redundant formulae are simply ignored. The other rules are expressed in the form most suitable for backward proof; exchange and contraction rules are not normally required, although they are provided anyway.

Figure 2.5 presents derived rules, including rules for $\leftrightarrow$. The weakened quantifier rules discard each quantification after a single use; in an automatic proof procedure, they guarantee termination, but are incomplete. Multiple use of a quantifier can be obtained by a contraction rule, which in backward proof duplicates a formula. The tactic `res_inst_tac` can instantiate the variable `?P` in these rules, specifying the formula to duplicate. See theory `Sequents/LK0` in the sources for complete listings of the rules and derived rules.

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

2.2 Automatic Proof

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

For classical reasoning, several tactics are available:

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

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

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

To control the set of rules for individual invocations, lower-case versions of all these primitives are available. Sections 2.8 and 2.9 give full details.

2.3 Tactics for the cut rule

According to the cut-elimination theorem, the cut rule can be eliminated from proofs of sequents. But the rule is still essential. It can be used to structure a proof into lemmas, avoiding repeated proofs of the same formula. More importantly, the cut rule cannot be eliminated from derivations of rules.

For example, there is a trivial cut-free proof of the sequent $P \land Q \vdash Q \land P$. Noting this, we might want to derive a rule for swapping the conjuncts in a right-hand formula:

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

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

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

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

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

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

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

2.4 Tactics for sequents

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

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

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

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

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

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

2.5 A simple example of classical reasoning

The theorem $\vdash \exists y . \forall x . P(y) \rightarrow P(x)$ is a standard example of the classical treatment of the existential quantifier. Classical reasoning is easy using LK, as you can see by comparing this proof with the one given in the FOL manual [10]. From a logical point of view, the proofs are essentially the same; the key step here is to use `exR` rather than the weaker `exR_thin`.

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

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

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

Because LK is a sequent calculus, the formula $P(?x)$ does not become an assumption; instead, it moves to the left side. The resulting subgoal cannot be instantiated to a basic sequent: the bound variable x is not unifiable with the unknown $?x$.

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

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

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

Subgoal 1 seems to offer lots of possibilities. Actually the only useful step is instantiating $?x_7$ to $\lambda x . x$, transforming $?x_7(x)$ into x.

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

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

```
Goal "|- EX y. ALL x. P(y)-->P(x)";
  Level 0
   |- EX y. ALL x. P(y) --> P(x)
   1.  |- EX y. ALL x. P(y) --> P(x)
by (best_tac LK_dup_pack 1);
  Level 1
   |- EX y. ALL x. P(y) --> P(x)
  No subgoals!
```

2.6 A more complex proof

Many of Pelletier's test problems for theorem provers [12] can be solved automatically. Problem 39 concerns set theory, asserting that there is no Russell set — a set consisting of those sets that are not members of themselves:

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

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

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

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

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

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

The rule `iffL` says, if $P \leftrightarrow Q$ then P and Q are either both true or both false. It yields two subgoals.

```
by (resolve_tac [iffL] 1);
  Level 4
   |- ~ (EX x. ALL y. F(y,x) <-> ~ F(y,y))
   1. !!x.  |- F(?x2(x),x), ~ F(?x2(x),?x2(x))
   2. !!x. ~ F(?x2(x),?x2(x)), F(?x2(x),x) |-
```

We must instantiate $?x_2$, the shared unknown, to satisfy both subgoals. Beginning with subgoal 2, we move a negated formula to the left and create a basic sequent.

```
by (resolve_tac [notL] 2);
  Level 5
   |- ~ (EX x. ALL y. F(y,x) <-> ~ F(y,y))
   1. !!x.  |- F(?x2(x),x), ~ F(?x2(x),?x2(x))
   2. !!x. F(?x2(x),x) |- F(?x2(x),?x2(x))
by (resolve_tac [basic] 2);
  Level 6
   |- ~ (EX x. ALL y. F(y,x) <-> ~ F(y,y))
   1. !!x.  |- F(x,x), ~ F(x,x)
```

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

```
by (resolve_tac [notR] 1);
  Level 7
   |- ~ (EX x. ALL y. F(y,x) <-> ~ F(y,y))
   1. !!x. F(x,x) |- F(x,x)
by (resolve_tac [basic] 1);
  Level 8
   |- ~ (EX x. ALL y. F(y,x) <-> ~ F(y,y))
  No subgoals!
```

2.7 *Unification for lists

Higher-order unification includes associative unification as a special case, by an encoding that involves function composition [5, page 37]. To represent lists, let C be a new constant. The empty list is $\lambda x\,.\,x$, while $[t_1, t_2, \ldots, t_n]$ is represented by

$$\lambda x\,.\,C(t_1, C(t_2, \ldots, C(t_n, x))).$$

The unifiers of this with $\lambda x\,.\,?f(?g(x))$ give all the ways of expressing $[t_1, t_2, \ldots, t_n]$ as the concatenation of two lists.

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

This technique lets Isabelle formalize sequent calculus rules, where the comma is the associative operator:

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

Multiple unifiers occur whenever this is resolved against a goal containing more than one conjunction on the left.

LK exploits this representation of lists. As an alternative, the sequent calculus can be formalized using an ordinary representation of lists, with a logic program for removing a formula from a list. Amy Felty has applied this technique using the language λProlog [3].

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

2.8 *Packaging sequent rules

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

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

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

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

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

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

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

`pack_of` *thy* returns the pack attached to theory *thy*.

`empty_pack` is the empty pack.

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

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

`LK_dup_pack` extends `prop_pack` with the safe rules `allR` and `exL` and the unsafe rules `allL` and `exR`. Search using this is complete, since quantified formulae may be reused, but frequently fails to terminate. It is generally unsuitable for depth-first search.

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

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

2.9 *Proof procedures

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

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

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

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

2.9.1 Method A

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

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

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

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

`repeat_goal_tac` *pack* i applies the safe rules in the pack to a goal and the resulting subgoals. If no safe rule is applicable then it applies an unsafe rule and continues.

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

2.9.2 Method B

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

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

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

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

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

`best_tac` *pack* i applies `step_tac` using best-first search to solve subgoal i. It is particularly useful for quantifier duplication (using `LK_dup_pack`).

Chapter 3

Defining A Sequent-Based Logic

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

3.1 Concrete syntax of sequences

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

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

- A sequence variable is any alphanumeric string with the first character being a `$` sign. So, consider the sequent `$A |- B`, where `$A` is intended to match a sequence of zero or more items.
- A sequence with unspecified sub-sequences and unspecified or individual items is written as a comma-separated list of regular variables (representing items), particular items, and sequence variables, as in

  ```
  $A, B, C, $D(x) |- E
  ```

  Here both `$A` and `$D(x)` are allowed to match any subsequences of items on either side of the two items that match B and C. Moreover, the sequence matching `$D(x)` may contain occurrences of x.
- An empty sequence can be represented by a blank space, as in ` |- true`.

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

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

3.2 Basis

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

A sequence can be represented as a function — a constructor for further sequences — by defining a binary *abstract* function `SeqO'` with type `[o,seq']=>seq'`, and translating a sequence such as `A, B, C` into

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

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

Furthermore, if we want to represent the sequence `A, $B, C`, we note that `$B` already represents a sequence, so we can use `B` itself to refer to the function, and therefore the sequence can be mapped to the internal form: `%s. SeqO'(A, B(SeqO'(C, s)))`.

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

3.3 Object logics

Recall that object logics are defined by mapping elements of particular types to the Isabelle type `prop`, usually with a function called `Trueprop`. So, an object logic proposition `P` is matched to the Isabelle proposition `Trueprop(P)`.

The name of the function is often hidden, so the user just sees `P`. Isabelle is eager to make types match, so it inserts `Trueprop` automatically when an object of type `prop` is expected. This mechanism can be observed in most of the object logics which are direct descendants of `Pure`.

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

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

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

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

```
two_seqi = [seq'=>seq', seq'=>seq'] => prop
two_seqe = [seq, seq] => prop
```

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

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

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

3.4 What's in `Sequents.thy`

Theory `Sequents.thy` makes many declarations that you need to know about:

1. The Isabelle types given below, which can be used for the constants that map object-level sequents and meta-level propositions:

```
single_seqe = [seq,seqobj] => prop
single_seqi = [seq'=>seq',seq'=>seq'] => prop
two_seqi    = [seq'=>seq', seq'=>seq'] => prop
two_seqe    = [seq, seq] => prop
three_seqi  = [seq'=>seq', seq'=>seq', seq'=>seq'] => prop
three_seqe  = [seq, seq, seq] => prop
four_seqi   = [seq'=>seq', seq'=>seq', seq'=>seq', seq'=>seq'] => prop
four_seqe   = [seq, seq, seq, seq] => prop
```

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

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

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

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

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

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

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

Chapter 4

Constructive Type Theory

Martin-Löf's Constructive Type Theory [7, 9] can be viewed at many different levels. It is a formal system that embodies the principles of intuitionistic mathematics; it embodies the interpretation of propositions as types; it is a vehicle for deriving programs from proofs.

Thompson's book [14] gives a readable and thorough account of Type Theory. Nuprl is an elaborate implementation [2]. ALF is a more recent tool that allows proof terms to be edited directly [6].

Isabelle's original formulation of Type Theory was a kind of sequent calculus, following Martin-Löf [7]. It included rules for building the context, namely variable bindings with their types. A typical judgement was

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

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

The theory `CTT` implements Constructive Type Theory, using natural deduction. The judgement above is expressed using $\bigwedge$ and $\Longrightarrow$:

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

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

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

To justify the CTT formulation it is probably best to appeal directly to the semantic explanations of the rules [7], rather than to the rules themselves. The order of assumptions no longer matters, unlike in standard Type Theory. Contexts, which are typical of many modern type theories, are difficult to represent in Isabelle. In particular, it is difficult to enforce that all the variables in a context are distinct.

The theory does not use polymorphism. Terms in CTT have type i, the type of individuals. Types in CTT have type t.

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

Figure 4.1: The constants of CTT

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

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

4.1 Syntax

The constants are shown in Fig. 4.1. The infixes include the function application operator (sometimes called 'apply'), and the 2-place type operators. Note that meta-level abstraction and application, $\lambda x . b$ and $f(a)$, differ from object-level abstraction and application, `lam` x`.` b and b`‘`a. A CTT function f is simply an individual as far as Isabelle is concerned: its Isabelle type is i, not say $i \Rightarrow i$.

The notation for CTT (Fig. 4.2) is based on that of Nordström et al. [9]. The empty type is called F and the one-element type is T; other finite types are built as $T + T + T$, etc.

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

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

The special cases as A`*`B and A`-->`B abbreviate general sums and products over a constant family.[1] Isabelle accepts these abbreviations in parsing and uses them whenever possible for printing.

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

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

BINDERS

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

INFIXES

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

TRANSLATIONS

prop	=	*type* `type`
	|	*type* `=` *type*
	|	*term* `:` *type*
	|	*term* `=` *term* `:` *type*
type	=	expression of type t
	|	`PROD` *id* `:` *type* `.` *type*
	|	`SUM` *id* `:` *type* `.` *type*
term	=	expression of type i
	|	`lam` *id* id^* `.` *term*
	|	`<` *term* `,` *term* `>`

GRAMMAR

Figure 4.2: Syntax of CTT

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

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

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

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

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

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

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

Figure 4.3: General equality rules

```
NF        N type

NI0       0 : N
NI_succ   a : N ==> succ(a) : N
NI_succL  a = b : N ==> succ(a) = succ(b) : N

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

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

NC0       [| a: C(0);
             !!u v. [| u: N; v: C(u) |] ==> b(u,v): C(succ(u))
          |] ==> rec(0, a, %u v. b(u,v)) = a : C(0)

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

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

Figure 4.4: Rules for type N

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

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

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

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

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

Figure 4.5: Rules for the product type $\prod_{x \in A} B[x]$

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

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

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

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

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

fst_def   fst(a) == split(a, %x y. x)
snd_def   snd(a) == split(a, %x y. y)
```

Figure 4.6: Rules for the sum type $\sum_{x\in A} B[x]$

4.2 Rules of inference

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

Figure 4.3 presents the equality rules. Most of them are straightforward: reflexivity, symmetry, transitivity and substitution. The judgement `Reduce` does not belong to Type Theory proper; it has been added to implement rewriting. The judgement `Reduce`(a, b) holds when $a = b : A$ holds. It also holds when a and b are syntactically identical, even if they are ill-typed, because rule `refl_red` does not verify that a belongs to A.

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

Figure 4.4 presents the rules for N, the type of natural numbers. They include `zero_ne_succ`, which asserts $0 \neq n + 1$. This is the fourth Peano

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

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

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

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

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

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

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

Figure 4.7: Rules for the binary sum type $A + B$

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

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

Figure 4.8: Rules for types F and T

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

Figure 4.9: Rules for the equality type $Eq(A, a, b)$

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

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

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

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

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

Figure 4.10: Derived rules for CTT

axiom and cannot be derived without universes [7, page 91].

The constant `rec` constructs proof terms when mathematical induction, rule `NE`, is applied. It can also express primitive recursion. Since `rec` can be applied to higher-order functions, it can even express Ackermann's function, which is not primitive recursive [14, page 104].

Figure 4.5 shows the rules for general product types, which include function types as a special case. The rules correspond to the predicate calculus rules for universal quantifiers and implication. They also permit reasoning about functions, with the rules of a typed λ-calculus.

Figure 4.6 shows the rules for general sum types, which include binary product types as a special case. The rules correspond to the predicate calculus rules for existential quantifiers and conjunction. They also permit reasoning about ordered pairs, with the projections `fst` and `snd`.

Figure 4.7 shows the rules for binary sum types. They correspond to the predicate calculus rules for disjunction. They also permit reasoning about disjoint sums, with the injections `inl` and `inr` and case analysis operator `when`.

Figure 4.8 shows the rules for the empty and unit types, F and T. They correspond to the predicate calculus rules for absurdity and truth.

Figure 4.9 shows the rules for equality types. If $a = b \in A$ is provable then `eq` is a canonical element of the type $Eq(A, a, b)$, and vice versa. These rules define extensional equality; the most recent versions of Type Theory use intensional equality [9].

Figure 4.10 presents the derived rules. The rule `subst_prodE` is derived from `prodE`, and is easier to use in backwards proof. The rules `SumE_fst` and `SumE_snd` express the typing of `fst` and `snd`; together, they are roughly equivalent to `SumE` with the advantage of creating no parameters. Section 4.12 below demonstrates these rules in a proof of the Axiom of Choice.

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

4.3 Rule lists

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

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

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

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

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

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

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

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

`basic_defs` contains the definitions of `fst` and `snd`.

4.4 Tactics for subgoal reordering

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

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

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

`typechk_tac` *thms* uses *thms* with formation, introduction, and elimination rules to check the typing of constructions. It is designed to solve goals of the form $a \in ?A$, where a is rigid and $?A$ is flexible; thus it performs type inference. The tactic can also solve goals of the form A type.

`equal_tac` *thms* uses *thms* with the long introduction and elimination rules to solve goals of the form $a = b \in A$, where a is rigid. It is intended for deriving the long rules for defined constants such as the arithmetic operators. The tactic can also perform type-checking.

`intr_tac` *thms* uses *thms* with the introduction rules to break down a type. It is designed for goals like $?a \in A$ where $?a$ is flexible and A rigid. These typically arise when trying to prove a proposition A, expressed as a type.

4.5 Rewriting tactics

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

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

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

include the computation rules and other equations. The long versions of the other rules permit rewriting of subterms and subtypes. Also used are transitivity and the extra judgement form `Reduce`. Meta-level simplification handles only definitional equality.

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

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

4.6 Tactics for logical reasoning

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

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

Isabelle provides several tactics for predicate calculus reasoning in CTT:

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

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

`mp_tac` i searches in subgoal i for assumptions of the form $f \in \Pi(A, B)$ and $a \in A$, where A may be found by unification. It replaces $f \in \Pi(A, B)$ by $z \in B(a)$, where z is a new parameter. The tactic can produce multiple outcomes for each suitable pair of assumptions. In short, `mp_tac` performs Modus Ponens among the assumptions.

`add_mp_tac` *i* is like `mp_tac` *i* but retains the assumption $f \in \Pi(A,B)$. It avoids information loss but obviously loops if repeated.

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

`safe_tac` *thms i* attempts to solve subgoal *i* by means of backtracking, using `safestep_tac`.

`step_tac` *thms i* tries to reduce subgoal *i* using `safestep_tac`, then tries unsafe rules. It may produce multiple outcomes.

`pc_tac` *thms i* tries to solve subgoal *i* by backtracking, using `step_tac`.

4.7 A theory of arithmetic

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

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

Figure 4.11 presents the definitions and some of the key theorems, including commutative, distributive, and associative laws.

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

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

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

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

4.8 The examples directory

This directory contains examples and experimental proofs in CTT.

`CTT/ex/typechk.ML` contains simple examples of type-checking and type deduction.

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

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

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

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

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

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

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

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

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

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

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

Figure 4.11: The theory of arithmetic

`CTT/ex/elim.ML` contains some examples from Martin-Löf [7], proved using `pc_tac`.

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

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

4.9 Example: type inference

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

```
Goal "lam n. rec(n, 0, %x y. x) : ?A";
  Level 0
  lam n. rec(n,0,%x y. x) : ?A
   1. lam n. rec(n,0,%x y. x) : ?A
```

Since the term is a Constructive Type Theory λ-abstraction (not to be confused with a meta-level abstraction), we apply the rule `ProdI`, for Π-introduction. This instantiates $?A$ to a product type of unknown domain and range.

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

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

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

Subgoal 1 is no longer flexible: we now know $?A_1$ is the type of natural numbers. However, let us continue proving nontrivial subgoals. Subgoal 2 asks, what is the type of 0?

```
by (resolve_tac [NI0] 2);
  Level 3
  lam n. rec(n,0,%x y. x) : N --> N
   1. N type
   2. !!n x y. [| x : N; y : N |] ==> x : N
```

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

```
by (assume_tac 2);
  Level 4
  lam n. rec(n,0,%x y. x) : N --> N
   1. N type
by (resolve_tac [NF] 1);
  Level 5
  lam n. rec(n,0,%x y. x) : N --> N
  No subgoals!
```

Calling `typechk_tac` can prove this theorem in one step.

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

```
Goal "lam n. rec(n, 0, %x y. tt) : ?A";
```

4.10 An example of logical reasoning

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

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

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

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

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

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

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

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

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

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

```
cut_facts_tac prems 1;
```

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

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

The subgoal has two new parameters, x and y. In the main goal, $?a$ has been instantiated with a `split` term. The assumption $y \in B(x) + C(x)$ is eliminated next, causing a case split and creating the parameter xa. This inference also inserts `when` into the main goal.

```
by (eresolve_tac [PlusE] 1);
  Level 2
  split(p,%x y. when(y,?c2(x,y),?d2(x,y)))
  : (SUM x:A. B(x)) + (SUM x:A. C(x))
   1. !!x y xa.
         [| x : A; xa : B(x) |] ==>
         ?c2(x,y,xa) : (SUM x:A. B(x)) + (SUM x:A. C(x))
   2. !!x y ya.
         [| x : A; ya : C(x) |] ==>
         ?d2(x,y,ya) : (SUM x:A. B(x)) + (SUM x:A. C(x))
```

To complete the proof object for the main goal, we need to instantiate the terms $?c_2(x, y, xa)$ and $?d_2(x, y, xa)$. We attack subgoal 1 by a +-introduction rule; since the goal assumes $xa \in B(x)$, we take the left injection (`inl`).

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

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

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

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

```
by (assume_tac 1);
  Level 5
  split(p,%x y. when(y,%xa. inl(<x,?b4(x,y,xa)>),?d2(x,y)))
  : (SUM x:A. B(x)) + (SUM x:A. C(x))
```

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

Subgoal 1 is an example of a well-formedness subgoal [2]. Such subgoals are usually trivial; this one yields to `typechk_tac`, given the current list of premises.

```
by (typechk_tac prems);
  Level 7
  split(p,%x y. when(y,%xa. inl(<x,xa>),?d2(x,y)))
  : (SUM x:A. B(x)) + (SUM x:A. C(x))
   1. !!x y ya.
         [| x : A; ya : C(x) |] ==>
         ?d2(x,y,ya) : (SUM x:A. B(x)) + (SUM x:A. C(x))
```

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

```
by (pc_tac prems 1);
  Level 8
  split(p,%x y. when(y,%xa. inl(<x,xa>),%y. inr(<x,y>)))
  : (SUM x:A. B(x)) + (SUM x:A. C(x))
  No subgoals!
```

Calling `pc_tac` after the first Σ-elimination above also proves this theorem.

4.11 Example: deriving a currying functional

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

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

Let us generalize this to the dependent types Σ and Π. The functional takes a function f that maps $z : \Sigma(A, B)$ to $C(z)$; the resulting function maps $x \in A$ and $y \in B(x)$ to $C(\langle x, y\rangle)$.

Formally, there are three typing premises. A is a type; B is an A-indexed family of types; C is a family of types indexed by $\Sigma(A, B)$. The goal is expressed using `PROD f` to ensure that the parameter corresponding to the functional's argument is really called f; Isabelle echoes the type using `-->` because there is no explicit dependence upon f.

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

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

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

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

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

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

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

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

Calling `pc_tac` would have proved this theorem in one step; it can also prove an example by Martin-Löf, related to $\vee$-elimination [7, page 58].

4.12 Example: proving the Axiom of Choice

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

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

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

But a completely formal proof is hard to find. The rules can be applied in countless ways, yielding many higher-order unifiers. The proof can get bogged down in the details. But with a careful selection of derived rules (recall Fig. 4.10) and the type-checking tactics, we can prove the theorem in nine steps.

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

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

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

Subgoal 1 asks to find the choice function itself, taking $x \in A$ to some $?b_7(h,x) \in B(x)$. Subgoal 2 asks, given $x \in A$, for a proof object $?b_8(h,x)$ to witness that the choice function's argument and result lie in the relation C. This latter task will take up most of the proof.

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

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

```
by (assume_tac 1);
  Level 3
  lam x. <lam xa. fst(x ` xa),lam xa. ?b8(x,xa)>
  : (PROD x:A. SUM y:B(x). C(x,y)) -->
    (SUM f:PROD x:A. B(x). PROD x:A. C(x,f ` x))
   1. !!h x.
         [| h : PROD x:A. SUM y:B(x). C(x,y); x : A |] ==>
         ?b8(h,x) : C(x,(lam x. fst(h ` x)) ` x)
```

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

```
by (resolve_tac [replace_type] 1);
  Level 4
  lam x. <lam xa. fst(x ` xa),lam xa. ?b8(x,xa)>
  : (PROD x:A. SUM y:B(x). C(x,y)) -->
    (SUM f:PROD x:A. B(x). PROD x:A. C(x,f ` x))
```

```
 1. !!h x.
       [| h : PROD x:A. SUM y:B(x). C(x,y); x : A |] ==>
       C(x,(lam x. fst(h ` x)) ` x) = ?A13(h,x)
 2. !!h x.
       [| h : PROD x:A. SUM y:B(x). C(x,y); x : A |] ==>
       ?b8(h,x) : ?A13(h,x)
```

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

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

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

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

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

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

We are finally ready to compose `snd` with h.

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

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

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

It might be instructive to compare this proof with Martin-Löf's forward proof of the Axiom of Choice [7, page 50].

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Index