A Preliminary User's Manual for Isabelle

Lawrence C Paulson
Computer Laboratory
University of Cambridge

Abstract

The theorem prover *Isabelle* and several of its object-logics are described. Where other papers [15] have been concerned with theory, the emphasis here is completely practical: the operations, commands, data structures, and organization of Isabelle. This information could benefit both users of Isabelle and implementors of other systems.

Contents

1 Introduction

Although the theorem prover Isabelle is still far from finished, there are enough intrepid users to justify the effort of writing a manual. Before reading this, you should be familiar with the ideas behind Isabelle [13, 15]. The manual describes pure Isabelle and several object-logics: an intuitionistic natural deduction calculus, Constructive Type Theory, a classical sequent calculus, and ZF set theory. The syntax, rules, and proof procedures of each logic are described.

Isabelle implements a meta-logic through term operations such as higher-order unification and parsing/display functions. To perform inference there are meta-level rules, proof tactics, and interactive proof commands.

Each object-logic implements simple proof procedures. These are reasonably powerful (at least for interactive use), if not always complete or systematic. These proof procedures illustrate how someone can piece together something useful from Isabelle's standard tactics and tacticals. Each logic is distributed with many sample proofs, some of which are described below. Though future Isabelle users will doubtless improve on these proof procedures, the examples already show that Isabelle can prove interesting theorems in various logics.

A serious theorem prover for classical logic might use of Bibel's matrix methods. Wallen [20] has extended these to modal and intuitionistic logic. Sophisticated methods offer efficiency and completeness, but most work only for certain fixed inference systems. With Isabelle you often work in an evolving inference system, deriving rules as you go.

When reading this report you may want to have Isabelle's sources at hand, and should be familiar with Standard ML [21]. The report describes a good many implementation details. These are not just for implementors of other systems. As with LCF, effective use of Isabelle involves ML programming: you must at least be prepared to write simple functions.

The line counts of pure Isabelle and each logic are worth noting, but with a pinch of salt. Such statistics give at best a crude picture of code size. They include everything: comments, signature declarations, listings of object-rules, and proofs.

pure	`NJ`	`CTT`	`LK`	`set`
4300	500	1100	600	1100

What about the user interface? Isabelle runs on workstations under standard window software. It is easy to make a menu: put common commands in a window where you can pick them up and insert them into an Isabelle session. One day Isabelle should have a better interface. But the fanciest mouse and window system in the world avails nothing unless the logic suits our needs and the theorem prover is powerful. When comparing systems, don't just admire the windows; find out what kind of theorem can be proved for a given amount of sweat.

Isabelle was first distributed in 1986. The 1987 version (distributed until April 1988) was the first to use higher-order logic and $\bigwedge$-lifting. The current version includes limited polymorphism, $\Longrightarrow$-lifting, and natural deduction.

This report is far from complete. Please send me any comments or corrections.

Acknowledgements. Isabelle uses Dave Matthews's Standard ML compiler, which is called Poly/ML. Philippe de Groote wrote the first version of LK. Funding and equipment were provided by the SERC/Alvey grant GR/E0355.7. Philippe Noël and Brian Monahan made many comments on the paper.

2 Types and terms

Isabelle is based on the idea that proofs in many logics can be naturally represented in intuitionistic higher-order logic [15], henceforth called the *meta-logic*. In Isabelle, like in higher-order logic, the terms are those of the typed λ-calculus.

2.1 The ML type `typ`

Every term has a type. The type `typ` is defined as follows:

```
infixr 5 -->;
datatype typ = Ground of string
             | Poly of string
             | op --> of typ * typ;
```

A *ground* type has a name, represented by a string, as does a *polymorphic* type (or type variable). A *function* type has the form `S-->T`. Two types are equal if they have identical structure: ML's equality test is correct for types.

A term of type `S-->T` denotes a function: if applied to a term of type `S`, the resulting term has type `T`. A term should obey the type-checking rules of the typed λ-calculus. It is possible to construct ill-typed terms, but the meta-rules ensure that all terms in theorems are well-typed.

Functions of several arguments are expressed by currying. The operator `-->` associates to the right, so

```
S1-->(S2-->(S3-->T))
```

can be written `S1-->S2-->S3-->T`. There is an ML operator `--->` for writing this as `[S1,S2,S3]--->T`.

Example: suppose that `f` has type `S-->T-->S`, where `S` and `T` are distinct types, `a` has type S, and `b` has type `T`. Then `f(a)` has type `T-->S` and `f(a,b)` has type `S`, while `f(b)`, `f(a,a)`, and `f(a,b,a)` are ill-typed. Note that `f(a,b)` means `(f(a))(b)`.

Type variables permit ML-style type inference so that variables need not be declared. The following meta-connectives are polymorphic:

$$\begin{aligned} \bigwedge &: (\sigma \to \mathit{prop}) \to \mathit{prop} \\ \equiv &: \sigma \to \sigma \to \mathit{prop} \end{aligned}$$

Type variables may not appear in theorems. Polymorphic theorems would complicate higher-order unification [12]; a polymorphic higher-order logic treads dangerously close to inconsistency [4]. A typed object-logic can be represented by making the object-level typing rules explicit. See the section on Constructive Type Theory, which is the ultimate example of a typed logic.

2.2 The ML type `term`

There are six kinds of term.

```
type indexname = string*int;

infix 9 $;
datatype term =
    Const of string * typ
  | Free  of string * typ
  | Var   of indexname * typ
  | Bound of int
  | Abs   of string*typ*term
  | op $  of term*term;
```

A *constant* has a name and a type. Constants include connectives like & and ∀ (logical constants), as well as constants like 0 and succ. Other constants may be required to define the abstract syntax of a logic.

A *free variable* (or `Free`) has a name and a type.

A *scheme variable* (or `Var`) has an indexname and a type, where an indexname is a string paired with a non-negative index. A `Var` is logically the same as a free variable. It stands for a variable that may be instantiated during unification.

The `Var`s in a term can be systematically renamed by incrementing the indices. In PROLOG jargon these are 'logical variables' and they may be 'standardized apart'.

A *bound variable* has no name, only a number: de Bruijn's representation [2]. The number counts the number of lambdas, starting from zero, between an occurrence of the variable and the lambda that binds it. The representation prevents capture of bound variables, allowing a simple and quick substitution function. The type of a bound variable is stored with its binding lambda (an `Abs` node). For more information see de Bruijn [2] or look at the functions `incr_boundvars`, `subst_bounds`, `aconv`.

An *abstraction* stores the name and type of its bound variable, and its body. The name is used only for parsing and printing; it has no logical significance.

A *combination* consists of a term applied to another term. The constructor is the infix `$`, so the ML expression `t$u` constructs the combination of `t` with `u`.

Two terms are *α-convertible* if they are identical up to renaming of bound variables.

- Two constants, `Free`s, or `Var`s are α-convertible just when their names and types are equal. (Variables having the same name but different types are thus distinct. This confusing situation should be avoided!)
- Two bound variables are α-convertible just when they have the same number.
- Two abstractions are α-convertible just when their bodies are, and their bound variables have the same type.
- Two combinations are α-convertible just when the corresponding subterms are.

Terms are never compared for equality, only for α-convertibility. Resolution uses β- and η-conversion to keep terms in long (η-expanded) normal form.

Remark. Isabelle originally implemented Martin-Löf's theory of expressions, which is similar to the typed λ-calculus. Martin-Löf calls these types *arities* to distinguish them from the types of Intuitionistic Type Theory. His latest theory of expressions includes pairing and projections. Isabelle does not: the unification algorithm would become yet more complicated. An object-logic can include a constant of type $\alpha \to (\alpha \to \alpha)$ for building pairs of type α.

2.3 Basic declarations

Exceptions

Exceptions in Isabelle are mainly used to signal errors. An exception includes a string (the error message) and other data to identify the error.

```
    exception TYPE: string * typ list * term list
```

Signals errors involving types and terms.

```
    exception TERM_ERROR: string * term list
```

Signals errors involving terms.

The logical constants

The following ML identifiers concern the symbols of the meta-logic.

```
    Aprop: typ
```

This is the type of propositions.

```
    implies: term
```

This is the implication symbol.

```
    all: typ -> term
```

The term `all T` is the universal quantifier for type `T`.

```
    equals: typ -> term
```

The term `equals T` is the equality predicate for type `T`.

2.4 Operations

There are a number of basic functions on terms and types.

```
    op ---> : typ list * typ -> typ
```

Given types $[\tau_1, \ldots, \tau_n]$ and τ, forms the type $\tau_1 \rightarrow \cdots \rightarrow (\tau_n \rightarrow \tau)$.

```
    loose_bnos: term -> int list
```

Calling `loose_bnos t` returns the list of all 'loose' bound variable references. In particular, `Bound 0` is loose unless it is enclosed in an abstraction. Similarly `Bound 1` is loose unless it is enclosed in at least two abstractions; if enclosed in just one, the list will contain the number 0. A well-formed term does not contain any loose variables.

```
    type_of: term -> typ
```

Computes the type of the term. Raises exception `TYPE` unless combinations are well-typed.

```
    op aconv: term*term -> bool
```

Are the two terms α-convertible?

```
    incr_boundvars: int -> term -> term
```

This increments a term's 'loose' bound variables by a given offset, required when moving a subterm into a context where it is enclosed by a different number of lambdas.

```
    abstract_over: term*term -> term
```

For abstracting a term over a subterm v: replaces every occurrence of v by a `Bound` variable with the correct index.

```
    subst_bounds: term list * term -> term
```

Applying `subst_bounds` to $[u_{n-1}, \ldots, u_0]$ and t substitutes the u_i for loose bound variables in t. This achieves β-reduction of $u_{n-1} \cdots u_0$ into t, replacing `Bound i` with u_i. For $(\lambda xy.t)(u, v)$, the bound variable indices in t are $x : 1$ and $y : 0$. The appropriate call is `subst_bounds([v,u],t)`. Loose bound variables $\geq n$ are reduced by n to compensate for the disappearance of n lambdas.

```
    subst_term: (term*term)list -> term -> term
```

Simultaneous substitution for atomic terms in a term. An atomic term is a constant or any kind of variable.

```
    maxidx_of_term: term -> int
```

Computes the maximum index of all the `Vars` in a term. If there are no `Vars`, the result is -1.

```
    term_match: (term*term)list * term*term -> (term*term)list
```

Calling `term_match(vts,t,u)` instantiates `Vars` in `t` to match it with `u`. The resulting list of variable/term pairs extends `vts`, which is typically empty. First-order pattern matching is used to implement meta-level rewriting.

2.5 The representation of object-rules

The module `Logic` contains operations concerned with inference — especially, for constructing and destructing terms that represent object-rules.

```
    op occs: term*term -> bool
```

Does one term occur in the other? (This is a reflexive relation.)

```
    add_term_vars: term*term list -> term list
```

Accumulates the `Vars` in the term, suppressing duplicates. The second argument should be the list of `Vars` found so far.

```
    add_term_frees: term*term list -> term list
```

Accumulates the `Frees` in the term, suppressing duplicates. The second argument should be the list of `Frees` found so far.

```
    mk_equals: term*term -> term
```

Given t and u makes the term $t \equiv u$.

```
    dest_equals: term -> term*term
```

Given $t \equiv u$ returns the pair (t, u).

```
    list_implies: term list * term -> term
```

Given the pair $([\phi_1, \ldots, \phi_m], \phi)$ makes the term $\phi_1 \Longrightarrow \cdots \phi_m \Longrightarrow \phi$.

```
    strip_imp_prems: term -> term list
```

Given $\phi_1 \Longrightarrow \cdots \phi_m \Longrightarrow \phi$ returns the list $[\phi_1, \ldots, \phi_m]$.

```
    strip_imp_concl: term -> term
```

Given $\phi_1 \Longrightarrow \cdots \phi_m \Longrightarrow \phi$ returns the term ϕ.

```
    list_equals: (term*term)list * term -> term
```

For adding flex-flex constraints to an object-rule. Given $([(t_1, u_1), \ldots, (t_l, u_l)], \phi)$, makes the term $t_1 \equiv u_1 \Longrightarrow \cdots t_l \equiv u_l \Longrightarrow \phi$.

```
    strip_equals: (term*term)list * term -> (term*term)list * term
```

Given $([], t_1 \equiv u_1 \Longrightarrow \cdots t_l \equiv u_l \Longrightarrow \phi)$, returns $([(t_l, u_l), \ldots, (t_1, u_1)], \phi)$.

```
    rule_of: (term*term)list * term list * term -> term
```

Makes an object-rule: given the triple

$$([(t_1, u_1), \ldots, (t_l, u_l)], [\phi_1, \ldots, \phi_m], \phi)$$

returns the term $t_1 \equiv u_1 \Longrightarrow \cdots t_l \equiv u_l \Longrightarrow \phi_1 \Longrightarrow \cdots \phi_m \Longrightarrow \phi$

```
strip_horn: term -> (term*term)list * term list * term
```

Breaks an object-rule into its parts: given

$$t_1 \equiv u_1 \Longrightarrow \cdots t_l \equiv u_l \Longrightarrow \phi_1 \Longrightarrow \cdots \phi_m \Longrightarrow \phi$$

returns the triple $([(t_l, u_l), \ldots, (t_1, u_1)], [\phi_1, \ldots, \phi_m], \phi)$

```
strip_assums: term -> (term*int) list * (string*typ) list * term
```

Strips premises of a rule allowing a more general form, where $\bigwedge$ and $\Longrightarrow$ may be intermixed. This is typical of assumptions of a subgoal in natural deduction. Returns additional information about the number, names, and types of quantified variables. For more discussion of assumptions, see Section 6.2.

```
strip_prems: int * term list * term -> term list * term
```

For finding premise (or subgoal) i: given the triple $(i, [], \phi_1 \Longrightarrow \cdots \phi_i \Longrightarrow \phi)$ it returns another triple, $(\phi_i, [\phi_{i-1}, \ldots, \phi_1], \phi)$, where ϕ need not be atomic. If i is out of range then raises an exception.

3 Parsing and Printing

There is a lexical analyzer and recursive descent parser for the syntax of pure Isabelle. The parser is written as a functional: it calls a user-supplied parser whenever it encounters an unrecognized construct. This allows extending the pure syntax with notation for an object-logic. Printing functions display expressions on the screen. Like the parser, the printer is a functional that can be supplied a printing function for a logic.

The parser and printer suppress redundant parentheses in expressions containing infix operators. An infix has a precedence from 1 (lowest) to 9 (highest), and associates either left or right.

The module `Symtab` defines a polymorphic type `'a table`. These symbol tables have lookup, update, and similar operations. The parser and printer use a symbol table that contains names of the variables, constants, infixes, and delimiters (keywords). The table also holds types and precedences.

Remark. It is hoped that the parser will be replaced by a parser generator that accepts syntax declarations, or at least a more general notion of operator. A pretty printer is also necessary. One day, a display will use nice fonts.

3.1 Lexical analysis

The type `lexsy` handles the five kinds of lexical symbols. Substrings of the input are paired with these symbols.

```
datatype lexsy =
    ConstSy  of typ
  | InfixSy  of typ * opprec
  | IdentSy  of typ
  | VarSy    of int * typ
  | DelimSy;
```

An *alphabetic identifier* is a sequence of letters, underscores (`_`), or primes (`'`); it must begin with a letter and MAY NOT CONTAIN DIGITS.

A *symbolic identifier* is a sequence containing only the following characters:

```
 !  #  $  %  &  *  +  -  <  >  /  \  ^  :  .  ;  ~  @  =  |  `
```

Each constant and infix (`ConstSy, InfixSy`) is an identifier that has a single type. Variable identifiers (`IdentSy`) need not be declared, for Isabelle can usually infer their types. Declaring an `IdentSy` specifies the type when the identifier appears as a bound, free, or scheme variable. This type cannot be overridden.

A *delimiter* (`DelimSy`) may be any identifier, or one of the following standard delimiters:

```
 (    )    [    ]    {    }    [|    |]    ,    %(    !(
```

These can be used in the syntax of an object-logic. Observe that the various brackets may not be used in identifiers.

3.2 Syntax

Pure Isabelle provides a notation for the meta-logic, namely higher-order logic (Figure 1). There is shorthand for defining and using functions of several arguments; the scope of abstractions and quantifications extends fully to the right. An object-logic typically extends the pure syntax by allowing object-expressions (possibly quite complex) within `[|` and `|]`.

Constants and free variables are written by name. The syntax of a scheme variable consists of a question mark (`?`) the string, and the number, all run together. The number can be omitted if it is zero. Examples: `?A`, `?B3`, `?Ga325`.

A bound variable cannot be parsed directly. The body of an abstraction is parsed, and then abstracted over the `Free` variable of the given name. Similarly, an abstraction is printed as though the bound variable were replaced by a `Free` variable; the printer chooses a unique name. A loose bound variable is printed `B.`n and usually indicates a bug.

If `&` is an infix operator then t`&`u denotes the application of `&` to the arguments t and u. Standard ML's `op` notation refers to the infix itself — we can write `op&` instead of `(%(x,y)x&y)`. Parentheses control the grouping of infixes.

Examples:

abbreviated	*fully parenthesized*
`%(x)f(x)`	`%(x)(f(x))`
`%(x,y,z)?b34`	`%(x)(%(y)(%(z)?b34))`
`R(u,f(a,b))`	`(R(u))((f(a))(b))`
`%(x)P(x)&Q(y)`	`%(x)((op&)(P(x),Q(y)))`

3.3 Declarations

The module `Syntax` declares a number of identifiers. (This module is normally closed; the identifiers can be used by qualification.)

The following ML type synonyms are assumed below. They are not present in Isabelle because ML modules do not currently permit type synonyms.

```
type lexemes = (string*lexsy)list;
type 'a parser = lexemes -> 'a*lexemes;
type printer = term -> unit;
```

Predefined identifiers include the following.

```
    exception LEXERR: string * string list
```

Signals errors in lexical analysis. The string list usually returns the remaining characters.

```
    exception PARSERR: string * lexemes
```

Signals parsing errors, returning the remaining lexical symbols.

```
    lex_string: lexsy table * string -> lexemes
```

This function reads a string and returns a lexeme list.

```
    pure_tab: lexsy table
```

The pure symbol table reserves the constants of the meta-logic: $\Longrightarrow$, $\equiv$, and $\bigwedge$.

Isabelle	*standard*
`%(`x_1`, ..., `x_n`)` t	the abstraction $\lambda x_1 \ldots x_n . t$
t `(`u_1`, ..., `u_n`)`	the combination $t(u_1, \ldots, u_n)$
`!(`x_1`, ..., `x_n`)` t	the quantification $\bigwedge x_1 \ldots x_n . t$

Notation

Isabelle	*standard*
`==`	the equality operator, $\equiv$
`==>`	the implication sign, $\Longrightarrow$

Infixes

$$
\begin{array}{rcl}
\textit{term} & = & \textit{constant} \\
& | & \textit{free-variable} \\
& | & \underline{\texttt{?}}\ \textit{scheme-variable} \\
& | & \underline{\texttt{\%(}}\ \textit{bound-vars}\ \underline{\texttt{)}}\ \textit{term} \\
& | & \textit{term}\ \underline{\texttt{(}}\ \textit{arguments}\ \underline{\texttt{)}} \\
& | & \textit{term infix term} \\
& | & \underline{\texttt{!(}}\ \textit{bound-vars}\ \underline{\texttt{)}}\ \textit{term} \\
& | & \underline{\texttt{(}}\ \textit{term}\ \underline{\texttt{)}} \\
& | & \text{[Object-logics may add other cases.]}
\end{array}
$$

$$\textit{bound-vars} \quad = \quad \textit{variable}\ \{\ \underline{\texttt{,}}\ \textit{variable}\ \}$$

$$\textit{arguments} \quad = \quad \textit{term}\ \{\ \underline{\texttt{,}}\ \textit{term}\ \}$$

Syntactic definition

Figure 1: Syntax of pure Isabelle

Parsing

The parser is written using functionals that operate on parsing functions to produce new ones. A parsing function, given a list of lexemes, returns some result paired with a tail of the lexemes. (Such techniques are well known in functional programming folklore.)

```
    parepeat: string * 'a parser -> ('a list)parser
```

Calls the parser function repeatedly (0 or more times), as long as the first symbol is the given delimiter. Returns the list of results.

```
    parepin: string * 'a parser -> ('a list)parser
```

Makes a parser function for

$$\textit{phrase}\ \underline{,}\ \ldots\ \underline{,}\ \textit{phrase}$$

where $\underline{,}$ can be any delimiter. Returns the list of results.

```
    parse_end: 'a * lexemes -> 'a
```

'Parses' the end of the list of lexemes — it checks that the list is empty, complaining if it is not.

```
    parse_functl: term parser -> int -> term parser
```

This is the functional for writing object-parsers. The integer gives precedence information about the context.

```
    op thenkey: 'a parser * string -> 'a parser
```

Parses a phrase, checks for the given keyword, and reads over it.

```
    op thenp: 'a parser * 'b parser -> ('a*'b)parser
```

Composition of functionals: parses one phrase, then another. Returns a pair of results.

Printing

There are several printing functions.

```
    string_of_type: typ -> string
```

Maps a type to a string, which can be printed.

```
    print_functl: (int*int->printer) * lexsy table ->
                  int*int->printer
```

This is the functional for writing object-printers. The integers give the precedence of the infixes to be printed on the left and right of the term.

```
    variant_abs: string * typ * term -> string * term
```

Given an abstraction, replaces the bound variable by a `Free` variable having a unique name.

4 Higher-order unification

Unification is used in the resolution of object-rules. Since logics are formalized in the typed λ-calculus, Isabelle uses Huet's higher-order unification algorithm [8].

4.1 Sequences

The module `Sequence` declares a type of unbounded sequences by the usual closure idea [14, page 118]. Sequences are defined in terms of the type `option`, declared in Isabelle's basic library, which handles the possible presence of a value.

```
datatype 'a option = None  |  Some of 'a;
```

Operations on the type `'a seq` include conversion between lists and sequences (with truncation), concatenation, and mapping a function over a sequence. Sequences are used in unification and tactics. The module `Sequence`, which is normally closed, declares the following.

```
    type 'a seq
```

The type of (possibly unbounded) sequences of type `'a`.

```
    seqof: (unit -> ('a * 'a seq) option) -> 'a seq
```

Calling `seqof (fn()=> Some(x,s))` constructs the sequence with head `x` and tail `s`, neither of which is evaluated.

```
    null: 'a seq
```

This is `seqof (fn()=> None)`, the empty sequence.

```
    single: 'a -> 'a seq
```

This is `seqof (fn()=> Some(x,null))`; makes a 1-element sequence.

```
    pull: 'a seq -> ('a * 'a seq) option
```

Calling `pull s` returns `None` if the sequence is empty and `Some(x,s')` if the sequence has head `x` and tail `s'`. Only now is `x` evaluated. (Calling `pull s` again will *recompute* this value! It is not stored!)

```
    append: 'a seq * 'a seq -> 'a seq
```

Concatenates two sequences.

```
    flats: 'a seq seq -> 'a seq
```

Concatenates a sequence of sequences.

```
    maps: ('a -> 'b) -> 'a seq -> 'b seq
```

Applies a function to every element of a sequence, producing a new sequence.

4.2 Environments

The module `Envir` (which is normally closed) declares a type of environments. An environment holds variable assignments and the next index to use when generating a variable.

```
    datatype env = Envir of {asol: term xolist, maxidx: int}
```

The operations of lookup, update, and generation of variables are used during unification.

```
    empty: int->env
```

Creates the environment with no assignments and the given index.

```
    lookup: env * indexname -> term option
```

Looks up a variable, specified by its indexname, and returns `None` or `Some` as appropriate.

```
    update: (indexname * term) * env -> env
```

Given a variable, term, and environment, produces *a new environment* where the variable has been updated. This has no side effect on the given environment.

```
    genvar: env * typ -> env * term
```

Generates a variable of the given type and returns it, paired with a new environment (with incremented `maxidx` field).

```
alist_of: env -> (indexname * term) list
```

Converts an environment into an association list containing the assignments.

```
norm_term: env -> term -> term
```

Copies a term, following assignments in the environment, and performing all possible β-reductions.

```
rewrite: (env * (term*term)list) -> term -> term
```

Rewrites a term using the given term pairs as rewrite rules. Assignments are ignored; the environment is used only with `genvar`, to generate unique `Var`s as placeholders for bound variables.

Remark. To minimize copying, earlier versions of Isabelle used environments throughout; structure sharing [1] was also tried. However, profiling showed that the majority of run-time was spent on lookups; also, the code was extremely complicated. Removing environments made Isabelle run 5 times faster. Structure sharing is becoming less popular in PROLOG interpreters as well.

Environments are represented by ordered association lists rather than by arrays. Poly/ML does not currently provide arrays. It is not clear that they could be used anyway: virtually all of the code is applicative.

4.3 The unification functions

The module `Unify` implements unification itself. It uses depth-first search with a depth limit that can be set. You can also switch tracing on and off, and specify a print function for tracing.

```
search_bound: int ref
```

Default 20, holds the depth limit for the unification search. The message

```
***Unification bound exceeded
```

appears whenever the search is cut off. This usually means the search would otherwise run forever, but a few proofs require increasing the default value of `search_bound`.

```
printer: (term->unit) ref
```

This function is used to print terms during tracing. It should be set to an object-logic's function `prin`. The default is a dummy that prints nothing.

```
    trace_bound: int ref
```

Default 10, tracing information is printed whenever the search depth exceeds this bound.

```
    trace_simp: bool ref
```

Default false, controls whether tracing information should include the SIMPL phase of unification. Otherwise only MATCH is traced.

```
    unifiers: env * ((term*term)list) -> (env * (term*term)list) seq
```

This is the main unification function. Given an environment and a list of disagreement pairs, it returns a sequence of outcomes. Each outcome consists of an updated environment and a list of flex-flex pairs (these are discussed below).

```
    smash_unifiers: env * (term*term)list -> env seq
```

This unification function maps an environment and a list of disagreement pairs to sequence of updated environments. The function obliterates flex-flex pairs by choosing the obvious unifier. It may be used to tidy up any flex-flex pairs remaining at the end of a proof.

Flexible-flexible disagreement pairs

A *flexible-flexible* disagreement pair is one where the heads of both terms are variables. Every set of flex-flex pairs has an obvious unifier and usually many others. The function `unifiers` returns the flex-flex pairs to constrain later unifications; `smash_unifiers` uses the obvious unifier to eliminate flex-flex pairs.

For example, the many unifiers of $?f(0) \equiv ?g(0)$ include $?f \mapsto \lambda x.?g(0)$ and $\{?f \mapsto \lambda x.x,\ ?g \mapsto \lambda x.0\}$. The trivial unifier, which introduces a new variable $?a$, is $\{?f \mapsto \lambda x.?a,\ ?g \mapsto \lambda x.?a\}$. Of these three unifiers, none is an instance of another.

Flex-flex pairs are simplified to eliminate redundant bound variables, as shown in the following example:

$$\lambda xy.?f(k(y), l(x)) \equiv \lambda xy.?g(y)$$

The bound variable x is not used in the right-hand term. Any unifier of these terms must delete all occurrences of x on the left. Choosing a new variable $?h$, the assignment $?f \mapsto \lambda uv.?h(u)$ reduces the disagreement pair to

$$\lambda xy.?h(k(y)) \equiv \lambda xy.?g(y)$$

without losing any unifiers. Now x can be dropped on both sides (adjusting bound variable indices) to leave

$$\lambda y.?h(k(y)) \equiv \lambda y.?g(y)$$

Assigning $?g \mapsto \lambda y.?h(k(y))$ eliminates $?g$ and unifies both terms to $\lambda y.?h(k(y))$.

Multiple unifiers

Higher-order unification can generate an unbounded sequence of unifiers. Multiple unifiers indicate ambiguity; usually the source of the ambiguity is obvious. Some unifiers are more natural than others. In solving $?f(a) \equiv a + b - a$, the solution $?f \mapsto \lambda x.x+b-x$ is better than $?f \mapsto \lambda x.a+b-a$ because it reveals the dependence of $a + b - a$ on a. There are four unifiers in this case. Isabelle generates the better ones first by preferring *projection* over *imitation*.

The unification procedure performs Huet's MATCH operation [8] in big steps. It solves $?f(t_1, \ldots, t_p) \equiv u$ for $?f$ by finding all ways of copying u, first trying projection on the arguments t_i. It never copies below any variable in u; instead it returns a new variable, resulting in a flex-flex disagreement pair. If it encounters $?f$ in u, it allows projection only. This prevents looping in some obvious cases, but can be fooled by cycles involving several disagreement pairs. It is also incomplete.

Associative unification

Associative unification comes for free: encoded through function composition, an associative operation [9, page 37]. To represent lists, let C be a new constant. The empty list is $\lambda x.x$, while the list $[t_1, t_2, \ldots, t_n]$ is represented by the term

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

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 standard associative unification, this technique can represent certain infinite sets of unifiers as finite sets containing flex-flex disagreement pairs. But $\lambda x.C(t, ?a)$ does not represent any list. Such garbage terms may appear in flex-flex pairs and accumulate dramatically.

Associative unification handles sequent calculus rules, where the comma is the associative operator:

$$\frac{\Gamma, A, B, \Delta \vdash \Theta}{\Gamma, A \,\&\, B, \Delta \vdash \Theta}$$

Multiple unifiers occur whenever this is resolved against a goal containing more than one conjunction on the left. Note that we do not really need associative unification, only associative *matching.*

5 Terms valid under a signature

The module `Sign` declares the abstract types of signatures and checked terms. A signature contains the syntactic information needed for building a theory. A checked term is simply a term that has been checked to conform with a given signature, and is packaged together with its type, etc.

5.1 The ML type `sg`

A signature lists all *ground types* that may appear in terms in the theory. The *lexical symbol table* declares each constant, infix, variable, and keyword. The *parsing and printing functions* implement the theory's notation. These functions have no logical meaning despite their practical importance.

```
datatype sg = Signat of
  {gnd_types: string list,
   lextab: lexsy Symtab.table,
   parser: term parser,
   printer: lexsy Symtab.table -> term -> unit,
   stamps: string ref list};
```

The *stamps* identify the theory. Each primitive theory has a single stamp. When the union of theories is taken, the lists of stamps are merged. References are used as the unique identifiers. The references are compared, not their contents.

Two signatures can be combined into a new one provided their *critical symbols* — constants, infixes, and delimiters — are compatible. If an identifier is used as a critical symbol in both signatures, it must be the same kind of symbol and have the same type in both signatures. This union operation should be idempotent, commutative, and associative. You can build signatures that ought to be the same but have different syntax functions, since functions cannot be compared.

5.2 The ML type `cterm`

A term t is *valid* under a signature provided every type in t is declared in the signature and every constant in t is declared as a constant or infix of the same type in the signature. It must be well-typed and monomorphic and must not have loose bound variables. Note that a subterm of a valid term need *not* be valid: it may contain loose bound variables. Even if $\lambda x.x$ is valid, its subterm x is a loose bound variable.

A checked term is stored with its signature, type, and maximum index of its Vars. This information is computed during the checks.

```
datatype cterm = Cterm of {sign: sg,
                           t: term,
                           T: typ,
                           maxidx: int};
```

The inference rules maintain that the terms that make up a theorem are valid under the theorem's signature. Rules (like specialization) that operate on terms take them as cterms rather than taking raw terms and checking them. It is possible to obtain cterms from theorems, saving the effort of checking the terms again.

5.3 Declarations

Here are the most important declarations of the module Sign. (This module is normally closed.)

```
type sg
```

The type of signatures, this is an abstract type: the constructor is not exported. A signature can only be created by calling new, typically via a call to prim_theory.

```
type cterm
```

The abstract type of checked terms. A cterm can be created by calling cterm_of or read_cterm.

```
rep_sg: sg -> {gnd_types: string list, lextab: lexsy table...}
```

The representation function for type sg, this returns the underlying record.

```
new: string -> ... -> sg
```

Calling new signame (gnd_types,lextab,parser,printer) creates a new signature named signame from the given type names, lexical table, parser, and printing functions.

```
rep_cterm: cterm -> {T: typ, maxidx: int, sign: sg, t: term}
```

The representation function for type cterm.

```
term_of: cterm -> term
```

Maps a cterm to the underlying term.

```
cterm_of: sg -> term -> cterm
```

Given a signature and term, checks that the term is valid in the signature and produces the corresponding cterm. Raises exception `TERM_ERROR` with the message 'type error in term' or 'term not in signature' if appropriate.

```
read_cterm: sg -> string*typ -> cterm
```

Reads a string as a term using the parsing information in the signature. It checks that this term is valid to produce a cterm. Note that a type must be supplied: this aids type inference considerably. Intended for interactive use, `read_cterm` catches the various exceptions that could arise and prints error messages. Commands like `goal` call `read_cterm`.

```
print_cterm: cterm -> unit
```

Prints the cterm using the printing function in its signature.

```
print_term: sg -> term -> unit
```

Prints the term using the printing function in the given signature.

```
type_assign: cterm -> cterm
```

Produces a cterm by updating the signature of its argument to include all variable/type assignments. Type inference under the resulting signature will assume the same type assignments as in the argument. This is used in the goal package to give persistence to type assignments within each proof. (Contrast with LCF's sticky types [14, page 148].)

6 Meta-inference

Theorems and theories are mutually recursive. Each theorem is associated with a theory; each theory contains axioms, which are theorems. To avoid circularity, a theorem contains a signature rather than a theory.

The module `Thm` declares theorems, theories, and all meta-rules. Together with `Sign` this module is critical to Isabelle's correctness: all other modules call on them to construct valid terms and theorems.

6.1 Theorems

The natural deduction system for the meta-logic [15] is represented by the obvious sequent calculus. Theorems have the form $\Phi \vdash \psi$, where Φ is the set of hypotheses and ψ is a proposition. Each meta-theorem has a signature and stores the maximum index of all the `Vars` in ψ.

```
datatype thm = Thm of
    {sign: Sign.sg,
     maxidx: int,
     hyps: term list,
     prop: term};
```

The proof state with subgoals $\phi_1, \ldots, \phi_m$ and main goal ϕ is represented by the object-rule $\phi_1 \ldots \phi_m/\phi$, which in turn is represented by the meta-theorem

$$\Phi \vdash t_1 \equiv u_1 \Longrightarrow \cdots t_l \equiv u_l \Longrightarrow \phi_1 \Longrightarrow \cdots \phi_m \Longrightarrow \phi \tag{1}$$

The field `hyps` holds Φ, the set of meta-level assumptions. The field `prop` holds the entire proposition, $t_1 \equiv u_1 \Longrightarrow \cdots \phi$, which can be further broken down. The function `tpairs_of` returns the (t, u) pairs, while `prems_of` returns the ϕ_i and `concl_of` returns ϕ.

```
    exception THM: string * int * thm list
```

Signals incorrect arguments to meta-rules. The tuple consists of a message, a premise number, and the premises.

```
    rep_thm: thm -> {prop: term, hyps: term list, ...}
```

This function returns the representation of a theorem, the underlying record.

```
    tpairs_of: thm -> (term*term)list
```

Maps the theorem (1) to the list of flex-flex constraints, $[(t_1, u_1), \ldots, (t_l, u_l)]$.

```
    prems_of: thm -> term list
```

Maps the theorem (1) to the premises, $[\phi_1, \ldots, \phi_m]$.

```
    concl_of: thm -> term
```

Maps the theorem (1) to the conclusion, ϕ.

Meta-rules

All of the meta-rules are implemented (though not all are used). They raise exception `THM` to signal malformed premises, incompatible signatures and similar errors.

```
    assume: Sign.cterm -> thm
```

Makes the sequent $\psi \vdash \psi$, checking that ψ contains no `Vars`. Recall that `Vars` are only allowed in the conclusion.

```
    implies_intr: Sign.cterm -> thm -> thm
```

This is $\Longrightarrow$-introduction.

```
    implies_elim: thm -> thm -> thm
```

This is $\Longrightarrow$-elimination.

```
    forall_intr: Sign.cterm -> thm -> thm
```

The $\bigwedge$-introduction rule generalizes over a variable, either `Free` or `Var`. The variable must not be free in the hypotheses; if it is a `Var` then there is nothing to check.

```
    forall_elim: Sign.cterm -> thm -> thm
```

This is $\bigwedge$-elimination.

```
    reflexive: Sign.cterm -> thm
```

Reflexivity of equality.

```
    symmetric: thm -> thm
```

Symmetry of equality.

```
    transitive: thm -> thm -> thm
```

Transitivity of equality.

```
    instantiate: (Sign.cterm*Sign.cterm) list -> thm -> thm
```

Simultaneous substitution of terms for distinct `Vars`. The result is not normalized.

Definitions

An axiom of the form $C \equiv t$ defines the constant C as the term t. Rewriting with the axiom $C \equiv t$ *unfolds* the constant C: replaces C by t. Rewriting with the theorem $t \equiv C$ (obtained by the rule `symmetric`) *folds* the constant C: replaces t by C. Several rules are concerned with definitions.

```
    rewrite_rule: thm list -> thm -> thm
```

This uses a list of equality theorems to rewrite another theorem. Rewriting is left-to-right and continues until no rewrites are applicable to any subterm.

```
    rewrite_goals_rule: thm list -> thm -> thm
```

This uses a list of equality theorems to rewrite just the antecedents of another theorem — typically a proof state. This unfolds definitions in the subgoals but not in the main goal.

Unfolding should only be needed for proving basic theorems about a defined symbol. Later proofs should treat the symbol as a primitive. For example, in first-order logic, bi-implication is defined in terms of implication and conjunction:

```
P<->Q == (P-->Q) & (Q-->P)
```

After deriving basic rules for this connective, we can forget its definition.

This treatment of definitions should be contrasted with many other theorem provers, where defined symbols are automatically unfolded.

6.2 Derived meta-rules for backwards proof

The following rules, coded directly in ML for efficiency, handle backwards proof. They typically involve a proof state

$$\psi_1 \Longrightarrow \cdots \psi_i \Longrightarrow \cdots (\psi_n \Longrightarrow \psi)$$

Subgoal i, namely ψ_i, might have the form

$$\bigwedge x_1.\theta_1 \Longrightarrow \cdots (\bigwedge x_k.\theta_k \Longrightarrow \theta)$$

Each θ_j may be preceded by zero or more quantifiers, whose scope extends to θ. The θ_j represent the *assumptions* of the subgoal; the x_j represent the *parameters.*

The object-rule $\phi_1 \Longrightarrow \cdots (\phi_m \Longrightarrow \phi)$ is lifted over the assumptions and parameters of the subgoal and renumbered [15]; write the lifted object-rule as

$$\widetilde{\phi}_1 \Longrightarrow \cdots (\widetilde{\phi}_m \Longrightarrow \widetilde{\phi})$$

Recall that, for example, $\widetilde{\phi}$ has the form

$$\bigwedge x_1.\theta_1 \Longrightarrow \cdots (\bigwedge x_k.\theta_k \Longrightarrow \phi')$$

where ϕ' is obtained from ϕ by replacing its free variables by certain terms.

Each rule raises exception THM if subgoal i does not exist.

```
resolution: thm * int * thm list -> thm Sequence.seq
```

Calling `resolution(state,i,rules)` performs higher-order resolution of a theorem in `rules`, typically an object-rule, with subgoal i of the proof state held in the theorem `state`. The sequence of theorems contains the result of each successful unification of $\widetilde{\phi} \equiv \psi_i$, replacing ψ_i by $\widetilde{\phi}_1, \ldots, \widetilde{\phi}_m$ and instantiating variables in `state`. The rules are used in order.

```
assumption: thm * int -> thm Sequence.seq
```

Calling `assumption(state,i)` attempts to solve subgoal i by assumption in natural deduction. The call tries each unification of the form $\theta_j \equiv \theta$ for $j = 1, \ldots, k$. The sequence of theorems contains the outcome of each successful unification, where ψ_i has been deleted and variables may have been instantiated elsewhere in the state.

```
biresolution: thm * int * (bool*thm)list -> thm Sequence.seq
```

Calling `biresolution(state,i,brules)` is like calling `resolution`, except that pairs of the form `(true,rule)` involve an implicit call of `assumption`. This permits using natural deduction object-rules in a sequent style, where the 'principal formula' is deleted after use. Write the lifted object-rule as

$$\widetilde{\phi}_1 \Longrightarrow \widetilde{\phi}_2 \Longrightarrow \cdots (\widetilde{\phi}_m \Longrightarrow \widetilde{\phi})$$

The rule is interpreted as an elimination rule with ϕ_1 as the major premise, and `biresolution` will insist on finding ϕ_1 among the assumptions of subgoal i. The call tries each unification of the form $\{\phi_1' \equiv \theta_j, \widetilde{\phi} \equiv \psi_i\}$ for $j = 1, \ldots, k$. The sequence of theorems contains the result of each successful unification, replacing ψ_i by the $m-1$ subgoals $\widetilde{\phi}_2, \ldots, \widetilde{\phi}_m$ and instantiating variables in `state`. The relevant assumption is deleted in the subgoals: if unification involved θ_j, then the occurrence of θ_j is deleted in each of $\widetilde{\phi}_2, \ldots, \widetilde{\phi}_m$.

Pairs of the form `(false,rule)` are treated exactly as in `resolution`. The pairs are used in order.

```
trivial: Sign.cterm -> thm
```

Makes the theorem $\psi \Longrightarrow \psi$, used as the initial state in a backwards proof. The proposition ψ may contain `Vars`.

6.3 Theories

The theory mechanism, while rudimentary, uses some concepts from Sannella and Burstall [17]. A *primitive* theory contains a signature and a set of named axioms. A *union* theory has two subtheories and includes their combined signature.

```
datatype theory =
    PrimThy of {sign: Sign.sg,  axioms: thm Symtab.table}
  | UnionThy of {sign: Sign.sg,  thy1: theory,  thy2: theory};
```

For examples of theory operations, see the theory `CTT` and its enrichments `arith` and `bool`. Another example is `set`, a theory built on top of `LK`.

```
get_axiom: theory -> string -> thm
```

Calling `get_axiom thy s` returns the axiom named `s` from the theory `thy` or its subtheories.

```
union_theory: theory * theory -> theory
```

Forms the union of two theories.

```
prim_theory: string ->
  (string list * lexsy table *
   term parser * (lexsy table->printer)) ->
  (string*string) list -> theory
```

Calling `prim_theory tn (...) axpairs` makes a primitive theory named `tn`, creating a new signature from the given type names, lexical table, and parsing/printing functions (recall Section 5.1). The `axpairs` are (name, axiom) pairs, where both names and axioms are strings. Each axiom is parsed under the new signature. Then all `Free`s are replaced by `Var`s — so that axioms can be written without question marks.

```
enrich_theory: theory -> string ->
  (string list * lexsy table *
   term parser * (lexsy table->printer)) ->
  (string*string)list -> theory
```

Calling `enrich_theory thy1 tn (...) axpairs` enriches `thy1` by adding ground types, updating the lexical table, providing new syntax functions, and adding axioms. The resulting theory, called `tn`, is the union of the new and old theories.

```
extend_theory: theory -> string ->
  (string list * lexsy) list -> (string*string) list -> theory
```

Calling `extend_theory thy1 tn lexdecs axpairs` extends `thy1` by adding entries to the lexical table and adding axioms. Each entry is a list of strings paired with a lexical class. This is a simple way to enrich a theory.

7 Tactics and tacticals

Isabelle tactics have little in common with LCF tactics apart from their purpose: to support backwards proof. An LCF tactic is essentially the inverse of an object-rule, mapping a goal to subgoals. An Isabelle tactic operates on a proof state, represented by an object-rule. It can solve several subgoals at once and even change the main goal (typically by instantiating variables).

An LCF proof state contains ML code that must be executed to achieve the goal; this code can fail, for an invalid tactic can return subgoals that do not imply the goal. An Isabelle proof state consists of a meta-theorem stating that the the subgoals imply the final goal. This represents the fringe of a goal tree; intermediate subgoals are discarded.

Resolution at the meta-level performs backwards proof at the object-level.

7.1 Derived rules

The module `Tactic` contains some derived rules, implemented using primitive rules.

```
    forall_intr_frees: thm -> thm
```

Generalizes a meta-theorem over all `Free` variables not free in hypotheses.

```
    forall_elim_vars: int -> thm -> thm
```

Replaces all outermost quantified variables by `Vars` of a given index.

```
    zero_var_indexes: thm -> thm
```

Replaces all `Vars` by `Vars` having index 0, preserving distinctness by renaming when necessary.

```
    standard: thm -> thm
```

Puts a meta-theorem into standard form: no hypotheses, `Free` variables, or outer quantifiers. All generality is expressed by `Vars` having index 0.

```
    resolve: thm * int * thm list -> thm
```

Calling `resolve (rlb,i,rules)` tries each of the `rules`, in turn, resolving them with premise i of `rlb`. Raises exception `THM` unless resolution produces exactly one result. This function can be used to paste object-rules together, making simple derived rules.

```
op RES: thm * thm -> thm
```

Calling `(rule2 RES rule1)` is equivalent to `resolve(rule2,1,[rule1])`; it resolves the conclusion of `rule1` with premise 1 in `rule2`. Raises exception `THM` unless there is exactly one unifier.

7.2 Printing functions

The module `Tactic` contains various printing functions.

```
print_thm: thm -> unit
```

Prints a meta-theorem.

```
print_thms: thm list -> unit
```

Prints a list of meta-theorems.

```
print_goal_thm: thm -> unit
```

Given the meta-theorem $\psi_1 \Longrightarrow \cdots\cdots (\psi_n \Longrightarrow \psi)$ prints ψ first, then the subgoals ψ_1, ..., ψ_n on separate numbered lines.

7.3 Tactics

An Isabelle tactic maps a proof state to a *sequence* of proof states, to allow the construction of a search tree. The proof search can produce a sequence (possibly infinite) of solutions to a goal.

```
datatype tactic = Tactic of thm -> thm Sequence.seq;
```

A tactic *fails* for a state if it returns the empty sequence; otherwise the tactic *succeeds*.

```
resolve_tac: thm list -> int -> tactic
```

The most fundamental tactic, `resolve_tac rules i` calls `resolution` with the given list of theorems and subgoal number. The given subgoal is matched against a list of object-rules, producing the sequence of all possible outcomes in order. The tactic fails if the goal number is out of range: thus with `REPEAT` (see below) it can often solve all subgoals $\geq i$:

```
REPEAT (resolve_tac rules i)
```

```
biresolve_tac: (bool*thm) list -> int -> tactic
```

This is analogous to `resolve_tac` but calls `biresolution`. It is thus suitable for a mixture of introduction and elimination rules.

```
assume_tac: int -> tactic
```

The tactic `assume_tac i` tries to solve subgoal i by assumption (in natural deduction). It calls the meta-rule `assumption`, which can return a sequence of results.

```
lift_inst_tac: (string*string*typ)list -> thm -> int -> tactic
```

For selective instantiation of variables, `lift_inst_tac [(v,t,T)] rule i` reads the strings `v` and `t`, obtaining a variable $?v$ and term t of the given type `T`. The term may refer to parameters of subgoal i, for the tactic modifies $?v$ and t to compensate for the parameters, and lifts the `rule` over the assumptions of the subgoal. The tactic replaces $?v$ by t in the rule and finally resolves it against the subgoal.

```
smash_all_ff_tac: tactic
```

Eliminates all flex-flex constraints from the proof state by applying the trivial higher-order unifier.

Meta-level rewriting

The following tactics fold and unfold definitions.

```
rewrite_tac: thm list -> tactic
```

Uses equality theorems to rewrite the proof state.

```
rewrite_goals_tac: thm list -> tactic
```

Uses equality theorems to rewrite the subgoals only.

```
fold_tac: thm list -> tactic
```

Uses equality theorems from right to left to rewrite the proof state, for folding definitions.

Identities of tacticals

These trivial tactics are used with tacticals.

```
all_tac: tactic
```

The identity element of the tactical `THEN`, `all_tac` maps any proof state to the 1-element sequence containing just that state. Thus it succeeds for all states.

```
no_tac: tactic
```

The identity element of the tacticals `ORELSE` and `APPEND`, `no_tac` maps any proof state to the empty sequence. Thus it succeeds for no state.

7.4 Filtering of object-rules

Higher-order resolution involving a long list of rules is slow. Filtering techniques can shorten the list of rules given to resolution. A second use of filtering is to detect whether resolving against a given subgoal would cause excessive branching. If too many rules are applicable then another subgoal might be selected.

The module `Stringtree` implements a data structure for fast selection of rules. A term is classified by its *head string*: the string of symbols obtained by following the first argument in function calls until a `Var` is encountered. For instance, the Constructive Type Theory judgement

```
rec(succ(?d),0,?b): N
```

has the head string `["Elem","rec","succ"]` where the constant `Elem` represents the elementhood judgement form $a \in A$.

Two head strings are *compatible* if they are equal or if one is a prefix of the other. If two terms have incompatible head strings, then they are clearly not unifiable. A theorem is classified by the head string of its conclusion, indicating which goals it could resolve with. This method is fast, easy to implement, and powerful.

Head strings are can only discriminate terms according to their first arguments. Type Theory introduction rules have conclusions like `0:N` and `inl(?a):?A+?B`. Because the type is the second argument, the head string does not discriminate by types.

```
could_resolve: term*term->bool
```

This function quickly detects nonunifiable terms. It assumes all variables are distinct, reporting that `?a=?a` may unify with `0=1`.

```
filt_resolve_tac: thm list -> int -> int -> tactic
```

Calling `filt_resolve_tac rules maxr i` uses `could_resolve` to discover which of the `rules` are applicable to subgoal `i`. If this number exceeds `maxr` then the tactic fails (returning the null sequence). Otherwise it behaves like `resolve_tac`.

```
compat_resolve_tac: thm list -> int -> int -> tactic
```

Calling `compat_resolve_tac rules maxr i` builds a stringtree from the `rules` to discover which of them are applicable to subgoal `i`. If this number exceeds `maxr` then the tactic fails (returning the null sequence). Otherwise it behaves like `resolve_tac`. (To avoid repeated construction of the same stringtree, apply `compat_resolve_tac` to a list of rules and bind the result to an identifier.)

7.5 Tacticals

Real proofs require hundreds of inferences. The module `Tactic` defines tacticals for various interesting search methods.

```
    op THEN: tactic * tactic -> tactic
```

The tactic `tac1 THEN tac2` applies `tac2` to each output from `tac1`, yielding sequential composition of tactics.

```
    op APPEND: tactic * tactic -> tactic
```

The tactic `tac1 APPEND tac2` returns the combined sequence containing the outputs of `tac1` followed by those of `tac2`, yielding nondeterministic choice of these two tactics.

```
    op ORELSE: tactic * tactic -> tactic
```

The tactic `tac1 ORELSE tac2` returns `tac1`'s output if this is non-empty; otherwise it returns `tac2`'s output. The choice of either `tac1` or `tac2` is deterministic. For example,

```
                REPEAT (tac1 ORELSE tac2)
```

produces a search tree where every branch uses `tac1` as much as possible before using `tac2`. It gives `tac1` priority over `tac2`, and causes less branching than `APPEND`.

```
    TRY: tactic -> tactic
```

Applying `TRY tac` to a state returns the output of `tac`, if non-empty, otherwise the original state. The definition is just

```
                fun TRY tac = tac ORELSE all_tac;
```

```
    REPEAT: tactic -> tactic
```

Repetition of a tactic is defined through `THEN` and `ORELSE`, like in LCF. Because `REPEAT` uses `ORELSE` rather than `APPEND`, every branch contains the greatest possible number of successful calls to the tactic.

```
ALLGOALS: (int -> tactic) -> tactic
```

Applying `ALLGOALS tf` to a proof state goes backwards through the subgoals applying `tf` to each one. (It goes backwards because `tf` may add or delete subgoals.) If there are n subgoals it behaves like

```
tf(n) THEN ... THEN tf(1)
```

```
SOMEGOAL: (int -> tactic) -> tactic
```

Applying `SOMEGOAL tf` to a proof state having n subgoals has the effect of

```
tf(n) ORELSE ... ORELSE tf(1)
```

```
DEPTH_FIRST: (thm -> bool) -> tactic -> tactic
```

Applying `DEPTH_FIRST sat tac` to a proof state applies `tac` in depth-first search to produce new proof states satisfying the predicate `sat`.

```
DEPTH_SOLVE_1: tactic -> tactic
```

Applying `DEPTH_SOLVE_1 tac` to a proof state applies `tac` in depth-first search to produce new proof states having fewer subgoals.

```
BREADTH_FIRST: (int -> thm -> bool) -> tactic -> tactic
```

Applying `BREADTH_FIRST sat tac` to a proof state applies `tac` in breadth-first search to produce new proof states satisfying the predicate `sat`. The first argument of `sat` is the depth; this can be used to abandon the search at a given depth.

```
DETERM: tactic -> tactic
```

The tactic `DETERM tac` is always deterministic. The output sequence from `tac` is chopped to delete all members after the first.

7.6 Deriving object-rules

To derive an object-rule, prove an object-theorem using the desired premises as meta-assumptions. The meta-rule `assume` makes such assumptions, and `standard` discharges them. The goal stack commands (below) use these operations to handle assumptions.

To derive a schematic object-rule, use `Free`s rather than `Var`s in the original goal. Schematic variables might become instantiated during the derivation. In first-order logic, if you are trying to derive $?P \,\&\, ?Q \supset ?Q \,\&\, ?P$, prove instead $P \,\&\, Q \supset Q \,\&\, P$ and then call `standard` to change the `Free` variables to `Var`s.

```
    prove_goal: theory -> string -> (thm list -> tactic list) -> thm
```

Calling `prove_goal thy a tacsf` derives the object-rule expressed by the string `a` in the theory `thy`. If the proposed object-rule is $\phi_1 \ldots \phi_m/\phi$ then the initial proof state is the meta-theorem $\vdash \phi \Longrightarrow \phi$. The premises are made into a list of meta-assumptions

$$\phi_1 \vdash \phi_1 \qquad \cdots \qquad \phi_m \vdash \phi_m$$

and given to the function `tacsf`. The resulting list of tactics are applied sequentially to produce a proof state. Then the meta-assumptions ϕ_1 up to ϕ_m are discharged. This theorem is returned (through `standard`) if it is $\vdash \phi_1 \Longrightarrow \cdots (\phi_m \Longrightarrow \phi)$; otherwise `prove_goal` reports an error.

This command is useful in batch files; see the derived set theory rules for many examples. Its syntax is similar to the `goal` command (see below) since proofs are usually found interactively, and later turned into batch proofs.

8 Goal stack package

The goal package facilitates interactive proof. It stores a current proof state and many previous states; commands can produce new states or return to previous ones. The *state list* at level n is a list of pairs

$$[(\psi_n, \Psi_n),\ (\psi_{n-1}, \Psi_{n-1}),\ \ldots,\ (\psi_0, [])]$$

where ψ_n is the current proof state, ψ_{n-1} is the previous one, ..., and ψ_0 is the initial proof state. The Ψ_i are sequences of proof states, storing branch points where a tactic returned a sequence longer than one.

Chopping elements from the state list reverts to previous proof states. Besides this, an `undo` command uses a list of previous states of the package itself.

```
    goal: theory -> string -> thm list
```

Calling `goal thy a` starts a proof. It reads the proposed object-rule from the string `a` in the theory `thy`. The conclusion yields the initial proof state. The premises are made into a list of assumptions and returned.

```
    by: tactic -> unit
```

Calling `by tac` applies a tactic to the proof state. If the tactic fails then an error occurs; otherwise its output sequence is non-empty. The head becomes the new proof state, and the tail is stored in the list.

```
pr: unit -> unit
```

Prints the top level proof state.

```
prlev: int -> unit
```

Calling `prlev n` prints the proof state at level `n`.

```
back: unit -> unit
```

Starting at the top level, `back` looks down the state list for an alternative state. The first one found becomes the current proof state. The previous state is discarded (but see `undo`), and the level is reset to that where the alternative was found.

```
chop: unit -> unit
```

Deletes the top level of the state list, canceling the effect of the last `by` command.

```
choplev: int -> unit
```

Calling `choplev n` truncates the state list to level `n`.

```
undo: unit -> unit
```

Undoes the last change to the proof state, including those caused by `chop` and `choplev` — but *not* those caused by `undo`. It can be repeated to the very start of the proof.

```
top_thm: unit -> thm
```

Returns the top level proof state.

```
result: unit -> thm
```

Returns the final result, applying `standard` to the proof state. Signals error unless the proof state has zero subgoals. The examples files call `result` after each proof to check that it succeeded.

```
get_goal: int -> term
```

Returns the given subgoal, say for debugging.

THIS ENDS THE DESCRIPTION OF PURE ISABELLE.
THE FOLLOWING SECTIONS DESCRIBE OBJECT-LOGICS.

9 Intuitionistic logic with natural deduction

The theory `NJ` implements intuitionistic first-order logic through Gentzen's natural deduction system, NJ. Natural deduction typically involves a combination of forwards and backwards reasoning, particularly with the rules &E, $\supset$E, and $\forall$E. Isabelle's backwards style does not handle these rules well, so the module `NJ/resolve` derives alternative rules. The result is an inference system similar to Gentzen's LJ (see Dummett [5, page 133]), a cut-free sequent calculus better suited to automatic proof. A simplistic proof procedure is provided.

This treatment of intuitionistic logic is descended from the theory `LJ`, an implementation, now withdrawn, of Gentzen's LJ. Natural deduction rules are simpler because of their implicit treatment of assumptions.

9.1 Syntax and rules of inference

The module `NJ_Syntax` defines the symbol table and syntax functions.

Figure 2 gives the lexical structure of `NJ`. The type of expressions is *exp* (written `Aexp` in ML), while the type of formulae is *form* (or `Aform`). The infixes are equality and the connectives. Note that `-->` is a constructor of the ML type `typ` as well as the implication sign for `NJ`.

Figure 3 gives the syntax for `NJ`. Quantifiers and negation get special treatment. Negation is defined through implication: `~P` expands to `P-->False`. The parser accepts `~P` as well as `~(P)`. Negation has a higher precedence than *all* infixes, so the parentheses in `~(a=b)` are necessary.

The module `NJ_Rule` declares meta-axioms (Figure 4) and binds them to ML identifiers of type `thm`. The connective $\leftrightarrow$ is defined using & and $\supset$. The equality rules are currently unused.

9.2 Derived rules and tactics

The module `NJ_Resolve` derives some rules (figure 5), which are used in sequent style. Giving `biresolve_tac` the pair `(true,disj_elim)` *replaces* an assumption $P \vee Q$ by two subgoals: one assuming P, the other assuming Q. In both subgoals the assumption $P \vee Q$ is clearly redundant.

The meta-theorem `asm_rl` is $[\![P]\!] \Longrightarrow [\![P]\!]$. (It really should be proved as $\phi \Longrightarrow \phi$ in the pure theory.) The pair `(true,asm_rl)` causes `biresolve_tac` to solve a goal by assumption. While `assume_tac` could perform the same function, this pair can be mixed with others in a proof procedure.

Let us call a rule *safe* if when applied to a provable goal the resulting subgoals will also be provable. If a rule is safe then it can be applied automatically to a goal without destroying our chances of finding a proof (provided the process terminates). All the rules of the classical sequent calculus LK are safe; not so for intuitionistic logic. The $\vee$I rules are obviously unsafe, for A can be false when $A \vee B$ is true.

The worst rule is implication elimination (`mp` or `imp_elim`). Given $P \supset Q$ we may assume Q provided we can prove P. If we were using classical logic then while proving P we could assume $\neg P$, but the intuitionistic proof of P may require repeated use of $P \supset Q$. And if the proof of P fails then the entire effort is wasted. Thus intuitionistic reasoning is hard even in propositional logic. For an elementary example, consider the NJ proof of Q from $P \supset Q$ and $(P \supset Q) \supset P$. The implication $P \supset Q$ is used twice.

$$\dfrac{P \supset Q \qquad \dfrac{(P \supset Q) \supset P \qquad P \supset Q}{P}}{Q}$$

The theorem prover for `NJ` does not use `imp_elim`; instead it uses derived rules to simplify each kind of implication (Figure 5). This simple technique is surprisingly powerful, but incomplete. Some of the derived rules are still unsafe but they process assumptions effectively. The idea is to reduce the antecedents of implications to atoms and then use Modus Ponens: from $P \supset Q$ and P deduce Q.

The tactic `mp_tac` performs Modus Ponens among the assumptions.

```
fun mp_tac i = biresolve_tac[(true,imp_elim)] i THEN assume_tac i;
```

This illustrates the power of tacticals, especially when the proof search branches. Calling `mp_tac i` calls `biresolve_tac`, which finds an implication $P \supset Q$ in the assumptions of subgoal i. It replaces that subgoal by two new subgoals numbered i and $i + 1$, where the new subgoal i is to prove P. The tactical `THEN` applies `assume_tac i`, which will only succeed if the new subgoal i has an assumption unifiable with P. Observe that `biresolve_tac` returns a *sequence* of proof states: one for every implication it finds. Then `assume_tac` removes those that lack the required assumption.

Another interesting tactic is `rebires_tac`. Suppose you wish to reduce subgoal i as much as possible using some set of rules. Then first try to solve the subgoal outright, trying those rules that produce no subgoals. Failing this, try to replace the subgoal by one new subgoal (trying the rules that produce one subgoal) and then recursively attack the new subgoal i. Finally try to replace the subgoal by two new subgoals; then recursively reduce the new subgoal $i + 1$ before turning to subgoal

i. If no rules apply, this procedure leaves the subgoal unchanged. It may not solve the given subgoal completely — there may be many new subgoals — but it will not disturb neighboring subgoals.[1] Contrast with the procedure of repeatedly applying rules to subgoal i. If it solves that goal then it will attack the former subgoal $i+1$.

So the NJ theorem prover consists of the following collection of tactics.

```
mp_tac: int -> tactic
```

If the given subgoal includes assumptions of the form $P \supset Q$ and P, the tactic replaces $P \supset Q$ by Q.

```
rebires_tac: (bool*thm) list -> int -> tactic
```

The basic idea is described above; it fails if it can do nothing. It also calls `mp_tac` before trying the 1-subgoal rules. Note that it uses `biresolve_tac`, giving sequent-style proof. An object-rule of n premises produces n subgoals if paired with `false`, but only $n-1$ subgoals if paired with `true`. Thus `(true,False_elim)` and `(true,asm_rl)` produce 0 subgoals.

```
safe_tac: int -> tactic
```

This repeatedly applies, using `rebires_tac`, the notionally safe rules. Some of them are really incomplete: quantified formulae are discarded after a single use. This tactic is mainly useful for demonstrations and debugging.

```
onestep_tac: int -> tactic
```

Exclusively for demonstrations, this tactic tries `safe_tac`, or else `mp_tac`, or else an unsafe step.

```
step_tac: int -> tactic
```

This is the basic unit for `pc_tac`. It repeatedly tries `safe_tac` and `mp_tac`, before considering an unsafe step.

```
pc_tac: int -> tactic
```

Uses `step_tac` in depth-first search to solve the given subgoal.

9.3 Examples

Examples distributed with NJ include proofs of dozens of theorems, such as these:

[1]The idea is due to Philippe de Groote.

```
[| (~ ~ P) & ~ ~ (P --> Q) --> (~ ~ Q) |]

[| (ALL x. ALL y. P(x) --> Q(y))    <->
   ((EXISTS x. P(x)) --> ALL y. Q(y))   |]

[| (EXISTS x. EXISTS y. P(x) & Q(x,y))
        <-> (EXISTS x. P(x) & EXISTS y. Q(x,y)) |]

[| (EXISTS y. ALL x. P(x) --> Q(x,y))
          --> ALL x. P(x) --> EXISTS y. Q(x,y) |]
```

Here is a session similar to one in my book [14, pages 222–3]. Compare the treatment of quantifiers in Isabelle and LCF. The output has been pretty printed by hand.

The proof begins by entering the goal. Then the rule ⊃I is used.

```
> goal NJ_Rule.thy "[| (EXISTS y. ALL x. Q(x,y))  \
# \              -->  (ALL x. EXISTS y. Q(x,y)) |]";
Level 0
[| (EXISTS y.ALL x.Q(x,y)) --> ALL x.EXISTS y.Q(x,y) |]
  1. [| (EXISTS y.ALL x.Q(x,y)) --> ALL x.EXISTS y.Q(x,y) |]
> by (resolve_tac [imp_intr] 1);
Level 1
[| (EXISTS y.ALL x.Q(x,y)) --> ALL x.EXISTS y.Q(x,y) |]
  1. [| EXISTS y.ALL x.Q(x,y) |] ==> [| ALL x.EXISTS y.Q(x,y) |]
```

The new state still has one subgoal, as do all the states in this example. The effect of ⊃I was to change `-->` into `==>`, so $\exists y.\forall x.Q(x,y)$ is now an assumption. The next step is ∀I.

```
> by (resolve_tac [all_intr] 1);
Level 2
[| (EXISTS y.ALL x.Q(x,y)) --> ALL x.EXISTS y.Q(x,y) |]
  1. [| EXISTS y.ALL x.Q(x,y) |] ==> !(ka)[| EXISTS y.Q(ka,y) |]
```

The effect of ⊃I was to replace the ∀-quantified variable `x` by the $\bigwedge$-quantified variable `ka`, called a *parameter* of the subgoal. The next step in the proof must be ∃I or ∃E. What happens if we choose the wrong rule?

```
> by (resolve_tac [exists_intr] 1);
```

```
Level 3
[| (EXISTS y.ALL x.Q(x,y)) --> ALL x.EXISTS y.Q(x,y) |]
  1. [| EXISTS y.ALL x.Q(x,y) |] ==> !(ka)[| Q(ka,?a2(ka)) |]
```

The new subgoal 1 contains the function variable `?a2`. Although `ka` is bound, a suitable assignment to `?a2` can replace the term `?a2(ka)` by a term containing `ka`. Now we simplify the assumption, $\exists y.\forall x.Q(x,y)$, using elimination rules. The first premise of ∃E can be any existential formula, but here it must be the assumption. To force this, we can compose `resolve_tac` with `assume_tac`.

```
> by (resolve_tac [exists_elim] 1 THEN assume_tac 1);
Level 4
[| (EXISTS y.ALL x.Q(x,y)) --> ALL x.EXISTS y.Q(x,y) |]
  1. [| EXISTS y.ALL x.Q(x,y) |] ==>
      !(ka,kb)[| ALL x.Q(x,kb) |] ==> [| Q(ka,?a2(ka)) |]
```

The `exists_elim` step has introduced the parameter `kb` and a new assumption. This is universally quantified: let us apply `all_elim` to it.

```
> by (resolve_tac [all_elim] 1 THEN assume_tac 1);
Level 5
[| (EXISTS y.ALL x.Q(x,y)) --> ALL x.EXISTS y.Q(x,y) |]
  1. [| EXISTS y.ALL x.Q(x,y) |] ==>
      !(ka,kb)[| ALL x.Q(x,kb) |] ==>
      [| Q(?a4(ka,kb),kb) |]      ==> [| Q(ka,?a2(ka)) |]
```

The subgoal now contains another function variable (`?a4`) and has three assumptions, one of which might be unifiable with the subgoal formula. Assigning `%(x,y)x` to `?a4` unifies `?a4(ka,kb)` with `ka`. But there is no way to unify `?a2(ka)` with the bound variable `kb`: assigning `%(x)kb` to `?a2` is illegal. (For more discussion see my other paper [15].)

```
> by (assume_tac 1);
by: tactic returned no results
Exception- ERROR with ... raised
```

And so `assume_tac` was unable to do anything with the subgoal. Using `choplev` we can return to the point where we made the wrong decision, and apply rules in the correct sequence.

```
> choplev 2;
```

```
Level 2
[| (EXISTS y.ALL x.Q(x,y)) --> ALL x.EXISTS y.Q(x,y) |]
  1. [| EXISTS y.ALL x.Q(x,y) |] ==> !(ka)[| EXISTS y.Q(ka,y) |]
> by (resolve_tac [exists_elim] 1 THEN assume_tac 1);
Level 3
[| (EXISTS y.ALL x.Q(x,y)) --> ALL x.EXISTS y.Q(x,y) |]
  1. [| EXISTS y.ALL x.Q(x,y) |] ==>
     !(ka,kb)[| ALL x.Q(x,kb) |] ==> [| EXISTS y.Q(ka,y) |]
```

We now have two parameters and no schematic variables. It is better to introduce parameters first since later `Vars` will be able to project onto them.

```
> by (resolve_tac [exists_intr] 1);
Level 4
[| (EXISTS y.ALL x.Q(x,y)) --> ALL x.EXISTS y.Q(x,y) |]
  1. [| EXISTS y.ALL x.Q(x,y) |] ==>
     !(ka,kb)[| ALL x.Q(x,kb) |] ==> [| Q(ka,?a3(ka,kb)) |]
> by (resolve_tac [all_elim] 1 THEN assume_tac 1);
Level 5
[| (EXISTS y.ALL x.Q(x,y)) --> ALL x.EXISTS y.Q(x,y) |]
  1. [| EXISTS y.ALL x.Q(x,y) |] ==>
     !(ka,kb)[| ALL x.Q(x,kb) |] ==>
     [| Q(?a4(ka,kb),kb) |] ==> [| Q(ka,?a3(ka,kb)) |]
```

The subgoal has variables `?a3` and `?a4` applied to both parameters. The obvious projection functions unify `?a4(ka,kb)` with `ka` and `?a3(ka,kb)` with `kb`.

```
> by (assume_tac 1);
Level 6
[| (EXISTS y.ALL x.Q(x,y)) --> ALL x.EXISTS y.Q(x,y) |]
```

Finally there are no subgoals. The theorem was proved in six tactic steps, not counting the abandoned ones. But proof checking is tedious: `pc_tac` proves the theorem in one step.

```
> choplev 0;
Level 0
[| (EXISTS y.ALL x.Q(x,y)) --> ALL x.EXISTS y.Q(x,y) |]
  1. [| (EXISTS y.ALL x.Q(x,y)) --> ALL x.EXISTS y.Q(x,y) |]
> by (pc_tac 1);
Level 1
[| (EXISTS y.ALL x.Q(x,y)) --> ALL x.EXISTS y.Q(x,y) |]
```

`Aexp`	individuals
`Aform`	truth values

Types

`.  ~  ALL  EXISTS`

Delimiters

symbol	*meta-type*	*precedence*	*description*
`=`	$[exp, exp] \rightarrow form$	Left 6	equality ($=$)
`&`	$[form, form] \rightarrow form$	Right 5	conjunction (&)
`|`	$[form, form] \rightarrow form$	Right 4	disjunction ($\vee$)
`-->`	$[form, form] \rightarrow form$	Right 3	implication ($\supset$)
`<->`	$[form, form] \rightarrow form$	Right 3	if and only if ($\leftrightarrow$)

Infixes

symbol	*meta-type*	*description*
`True`	$form \rightarrow prop$	meta-predicate of truth
`Forall`	$(exp \rightarrow form) \rightarrow form$	universal quantifier ($\forall$)
`Exists`	$(exp \rightarrow form) \rightarrow form$	existential quantifier ($\exists$)
`False`	$form$	absurd formula ($\perp$)

Constants

Figure 2: Lexical symbols for `NJ`

Isabelle notation	*expansion*	*standard notation*
`ALL` x`.` P	`Forall(`$\lambda x.P$`)`	universal quantification $\forall x.P$
`EXISTS` x`.` P	`Exists(`$\lambda x.P$`)`	existential quantification $\exists x.P$
`~`P	P`-->False`	the negation $\neg P$

NOTATION

$$\begin{array}{rcl} \textit{term} & = & \underline{\texttt{ALL}}\ \textit{var}\ \underline{\texttt{.}}\ \textit{term} \\ & \mid & \underline{\texttt{EXISTS}}\ \textit{var}\ \underline{\texttt{.}}\ \textit{term} \\ & \mid & \underline{\texttt{\~{}}}\ \textit{term} \\ & \mid & \underline{\texttt{[|}}\ \textit{term}\ \underline{\texttt{|]}} \\ & \mid & \textit{other Isabelle terms} \ldots \end{array}$$

SYNTACTIC DEFINITION

Figure 3: Syntax of `NJ`

```
refl        [| a=a |]
sym         [| a=b |] ==> [| b=a |]
trans       [| a=b |] ==> [| b=c |] ==> [| a=c |]
```

Equality rules

```
conj_intr   [| P |] ==> [| Q |] ==> [| P&Q |]
conjunct1   [| P&Q |] ==> [| P |]
conjunct2   [| P&Q |] ==> [| Q |]

disj_intr1  [| P |] ==> [| P|Q |]
disj_intr2  [| Q |] ==> [| P|Q |]

disj_elim   [| P|Q |] ==> ([| P |] ==> [| R |]) ==>
                          ([| Q |] ==> [| R |]) ==> [| R |]

imp_intr    ([| P |] ==> [| Q |]) ==> [| P-->Q |]
mp          [| P-->Q |] ==> [| P |] ==> [| Q |]

False_elim  [| False |] ==> [| P |]

iff_def     P<->Q == (P-->Q) & (Q-->P)
```

Propositional rules

```
all_intr    (!(u) [| P(u) |]) ==> [| Forall(P) |]
spec        [| Forall(P) |] ==> [| P(a) |]

exists_intr [| P(a) |] ==> [| Exists(P) |]
exists_elim [| Exists(P) |] ==>
            (!(u)[| P(u) |] ==> [| R |]) ==> [| R |]
```

Quantifier rules

Figure 4: Meta-axioms for NJ

```
conj_elim  [| P&Q |] ==> ([| P |] ==> [| Q |] ==> [| R |]) ==>
               [| R |]
imp_elim   [| P-->Q |] ==> [| P |] ==> ([| Q |] ==> [| R |]) ==>
               [| R |]
all_elim   [| Forall(P) |] ==> ([| P(a) |] ==> [| R |]) ==> [| R |]
```

Sequent-style elimination rules

```
iff_intr    ([| P |] ==> [| Q |]) ==> ([| Q |] ==> [| P |]) ==>
               [| P<->Q |]
iff_elim    [| P <-> Q |] ==>
            ([| P-->Q |] ==> [| Q-->P |] ==> [| R |]) ==> [| R |]
```

Natural deduction rules for ‘if and only if’

```
conj_imp_elim [| (P&Q)-->S |] ==> ([| P-->(Q-->S) |] ==> [| R |])
              ==> [| R |]
disj_imp_elim [| (P|Q)-->S |] ==>
              ([| P-->S |] ==> [| Q-->S |] ==> [| R |]) ==> [| R |]
imp_imp_elim  [| (P-->Q)-->S |] ==>
              ([| P |] ==> [| Q-->S |] ==> [| Q |]) ==>
              ([| S |] ==> [| R |]) ==>    [| R |]
iff_imp_elim  [| (P<->Q)-->S |] ==>
              ([| P |] ==> [| Q-->S |] ==> [| Q |]) ==>
              ([| Q |] ==> [| P-->S |] ==> [| P |]) ==>
              ([| S |] ==> [| R |]) ==>    [| R |]
all_imp_elim  [| Forall(P)-->S |] ==> (!(x)[| P(x) |]) ==>
              ([| S |] ==> [| R |]) ==>   [| R |]
exists_imp_elim
  [| Exists(P)-->S |] ==> ([| P(a)-->S |] ==> [| R |]) ==> [| R |]
```

Special-case implication rules

Figure 5: Derived rules for NJ

10 Constructive Type Theory

Isabelle was first written for the Intuitionistic Theory of Types, a formal system of great complexity.[2] The original formulation was a kind of sequent calculus with rules for building the context (variable:type bindings). A typical judgement was

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

In early releases of Isabelle, the object-logic `ITT` implemented this sequent calculus. It was not completely satisfactory, particularly for reasoning about arbitrary types and families. Natural assumptions like 'suppose A is a type' or 'suppose $B(x)$ is a type for all x in A' could not be formalized. Such problems led Martin-Löf to seek richer 'logical frameworks' [11] — and stimulated many people to think about generic theorem proving.

The new logical framework of Isabelle (namely, higher-order logic) permits a natural deduction formulation of Type Theory. The judgement above is expressed using $\bigwedge$ and $\Longrightarrow$:

$$\begin{aligned}\bigwedge x_1 . \llbracket x_1 \in A_1 \rrbracket \Longrightarrow \bigwedge x_2 . \llbracket x_2 \in A_2(x_1) \rrbracket \Longrightarrow \cdots \bigwedge x_n . \llbracket x_n \in A_n(x_1, \ldots, x_{n-1}) \rrbracket \\ \Longrightarrow \llbracket a(x_1, \ldots, x_n) \in A(x_1, \ldots, x_n) \rrbracket\end{aligned}$$

Now assumptions can use all the judgement forms, not just $x \in A$, and can even express that B is a family of types over A:

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

This Isabelle logic is called `CTT` (Constructive Type Theory) to distinguish it from its obsolete predecessor. To justify the `CTT` formulation it is probably best to appeal directly to the semantic explanations of the rules [10], rather than to the rules themselves. Note that the order of assumptions no longer matters, unlike in standard Type Theory. Frankly I am not sure how faithfully `CTT` reflects Martin-Löf's semantics; but many researchers using Type Theory do not bother with such considerations.

All of Type Theory is supported 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 of `Aexp` and `Atype`.

`CTT` uses the 1982 version of equality, where the judgements $a = b \in A$ and $c \in Eq(A, a, b)$ are interchangeable. Its rewriting tactics prove theorems of the form $a = b \in A$. Under the new equality rules, rewriting tactics would have to prove theorems of the form $c \in Eq(A, a, b)$, where c would be a large construction.

[2]This section presupposes knowledge of Martin-Löf [10].

10.1 Syntax and rules of inference

The module `CTT_Syntax` defines the symbol table and syntax functions.

Figure 6 gives the lexical symbols of `CTT`. There are meta-types of types and expressions. The constants are shown in Figure 7. The infixes include the function application operator (sometimes called 'apply'), and the 2-place type operators. The empty type is called F and the one-element type is T; other finite sets are built as $T+T+T$, etc.

The `CTT` syntax (Figure 8) is similar to that used at the University of Gothenburg, Sweden. We can write `SUM y:B. PROD x:A. C(x,y)` instead of

```
Sum(B,%(y)Prod(A,%(x)C(x,y)))
```

The module `CTT_Rule` binds the rules of Type Theory to ML identifiers. The equality versions of the rules are called *long* versions; the rules describing the computation of eliminators are called *computation* rules. Some rules are reproduced here to illustrate the syntax. Figure 9 shows the rules for $+$, the sum of two types. Figure 10 shows the rules for N. These include `zero_ne_succ`, the fourth Peano axiom ($0 \neq n+1$) because it cannot be derived without universes [10, page 91]. Figure 11 shows the rules for the general product.

The extra judgement 'reduce' is used to implement rewriting. The judgement reduce(a,b,A) 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. These rules do not give rise to new theorems about the standard judgements — note that the only rule that makes use of 'reduce' is `trans_red`, whose first premise ensures that a and b (and thus c) are well-typed.

Figure 12 shows the rules for 'reduce' and the definitions of $\rightarrow$, $\times$, fst, and snd. No special rules are defined for $\rightarrow$ and $\times$; their definitions should always be unfolded. (Perhaps the parser should do this.)

Many proof procedures work by repeatedly resolving certain Type Theory rules against a proof state. The module `CTT_Resolve` includes various useful lists of related rules.

```
form_rls: thm list
```

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

```
form_long_rls: thm list
```

These are the long formation rules for Π, Σ, $+$, and Eq. (For N, F, and T use `refl_type`.)

```
intr_rls: thm list
```

These are the introduction rules for the types N, Π, Σ, $+$, and T. There is no introduction rule for F.

```
intr_long_rls: thm list
```

These are the long introduction rules for N, Π, Σ, and $+$. (For T use `refl_elem`.)

```
elim_rls: thm list
```

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

```
elim_long_rls: thm list
```

These are the long elimination rules for N, Π, Σ, $+$, and F.

```
comp_rls: thm list
```

These are the computation rules for the types N, Π, Σ, and $+$. There is no computation rule for F, while those for Eq and T involve no eliminator.

```
basic_defs: thm list
```

These are the definitions shown in Figure 12. Rewriting with `basic_defs` unfolds them all.

10.2 Tactics

The module `CTT_Resolve` declares Type Theory tactics. Powerful procedures, including simplification, type inference, and logical reasoning, can solve many kinds of goals.

Derived rules are shown in Figure 13. The rule `subst_prod_elim` is derived from `prod_elim`, and is easier to use in backwards proof. The rules `Sum_elim_fst` and `Sum_elim_snd` express the intuitive properties of `fst` and `snd`.

Subgoal reordering

Blind application of rules seldom leads to a proof. Many rules, especially elimination rules, create subgoals containing new schematic variables. Such variables unify with anything, causing an undirectional search. The standard tactics `filt_resolve_tac` and `compat_resolve_tac` (Section 7.4) can reject ambiguous goals; so does the `CTT` tactic `test_assume_tac`. Used with the `CTT` tactical `REPEAT_FIRST` they achieve a simple kind of subgoal reordering.

```
test_assume_tac: int -> tactic
```

Calling `test_assume_tac i`, where subgoal i has the form $a \in A$ and the head of a is not a `Var`, calls `assume_tac` to solve the subgoal by assumption. All other cases are rejected.

```
REPEAT_FIRST: (int -> tactic) -> tactic
```

Applying `REPEAT_FIRST tf` searches for the least `i` for which `tf` succeeds, then repeatedly applies `tf(i)`. The entire process repeats until `tf` fails on all subgoals (ideally, because no subgoals are left!).

Repetitive tactics

A great many `CTT` tactics apply the operations above. The most important are the following.

```
typechk_tac: tactic
```

This tactic uses formation, introduction, and elimination rules to check the typing of constructions. It is designed to solve goals like $a \in {?A}$ where a is rigid and $?A$ is flexible. Thus it performs type inference using essentially Milner's algorithm, which is expressed in the rules. The tactic can also solve goals of the form A type.

```
equal_tac: tactic
```

This tactic is designed to solve goals like $a = b \in A$, where a is rigid, using the long introduction and elimination rules. It is intended for deriving the long rules for defined constants such as the arithmetic operators. The tactic can also solve goals of the form A type.

```
intr_tac: tactic
```

This tactic uses 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.

Simplification

Object-level simplification is accomplished through proof, using the `CTT` equality rules. The rewrites are the computation rules and the long versions of the other rules. Also used are transitivity and the extra judgement form 'reduce'. Meta-level simplification handles only definitional equality.

```
simp_tac: tactic
```

A simplifier nearly as powerful as LCF's, `simp_tac` applies left-to-right rewrites, solving the goal $a = b \in A$ by rewriting a to b. If b is a `Var` then it is assigned the rewritten form of a.

```
presimp_tac: tactic
```

This prepares for simplification by breaking each subgoal $a = b \in A$ into the two subgoals $b = ?c \in A$ and $a = ?c \in A$, where $?c$ is a new `Var`.

Logical reasoning

The interpretation of propositions as types allows `CTT` to express statements of intuitionistic logic. The proof procedures of `NJ`, adapted for `CTT`, can prove many such statements automatically.

The key question is, can we delete an assumption after using it with an elimination rule? The situation is subtler than it looks. 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. 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 $z \in A+B$ may render the subgoals unprovable if other assumptions refer to z. (Some people might argue that such subgoals are not even meaningful.)

```
mp_tac: int -> tactic
```

If the given subgoal contains assumptions $f \in \Pi(A, B)$ and $a \in A$, then this tactic adds the assumption that there is some $z \in B(a)$. The assumption $f \in \Pi(A, B)$ is kept.

```
bi_mp_tac: int -> tactic
```

Similar to `mp_tac`, but deletes the assumption $f \in \Pi(A, B)$.

```
rebires_tac: (bool * thm) list -> int -> tactic
```

Similar to its namesake in the theory `NJ`.

```
safe_tac: thm list -> int -> tactic
```

Calling `safe_tac asms i` attacks subgoal i using formation rules and certain other 'safe' rules (`F_elim`, `Prod_intr`, `Sum_elim`, `Plus_elim`), calling `bi_mp_tac` when appropriate. It also uses the theorems `asms`, which are typically meta-assumptions, premises of the rule being derived. (The name `safe_tac`, taken from `NJ`, is inappropriate for Type Theory. This tactic is certainly incomplete.)

```
unsafe_tac: int -> tactic
```

This attacks the subgoal with the 'unsafe' rules `Plus_intr_inl`, `Plus_intr_inr`, `Sum_intr`, `subst_prod_elim`.

```
step_tac: thm list -> int -> tactic
```

The basic unit for `solve_tac`, this tries `safe_tac` and `unsafe_tac` on the given subgoal.

```
solve_tac: thm list -> int -> tactic
```

Calling `solve_tac` on a subgoal performs a depth-first search using `step_tac` to solve it.

10.3 Examples files

The directory `CTT/ex` includes files of examples to demonstrate type checking, trivial forms of program synthesis, and simplification.

Examples of logical reasoning

The tactic `solve_tac` automatically solves many examples of logical reasoning.[3] Each goal is expressed using the variable `?a` in place of the desired construction (or proof object). In the course of the proof, Isabelle instantiates this variable.

A distributive law of $\times$ over $+$ generalizes to a meta-theorem about Σ. Observe the premises that A is a type and that B and C are families.

```
[| A type |] ==>
(!(x)[| x:A |] ==> [| B(x) type |]) ==>
(!(x)[| x:A |] ==> [| C(x) type |]) ==>
[| ?a: (SUM x:A.B(x)+C(x)) --> (SUM x:A.B(x))+(SUM x:A.C(x)) |]
```

By the end of the proof, `?a` has become

```
lam ka. split(ka,%(kb,kc)when(kc,%(kd)inl(<kb,kd>),
                                  %(kd)inr(<kb,kd>)))
```

[3]These are on the file `CTT/ex/elim.ML`

The following proof derives a currying functional in `?a`. The argument of the functional is a function that maps $z : \Sigma(A, B)$ to $C(z)$; the resulting function maps $x \in A$ and $y \in B(x)$ to $C(\langle x, y\rangle)$. Here B is a family over A while C is a family over $\Sigma(A, B)$.

```
[| A type |] ==>
(!(x)[| x:A |] ==> [| B(x) type |]) ==>
(!(z)[| z: (SUM x:A . B(x)) |] ==> [| C(z) type |]) ==>
[| ?a: (PROD z: (SUM x:A . B(x)) . C(z))
                  --> (PROD x:A . PROD y:B(x) . C(<x,y>)) |]
```

By the end of the proof, `?a` has become

```
lam ka. lam kb. lam kc. ka ‘ <kb,kc>
```

An example of Martin-Löf [10, page 58] is the axiom of $\vee$ elimination. This meta-theorem is an alternative form of the rule `Plus_elim`. Here C is a family over $A + B$.

```
[| A type |] ==> [| B type |] ==>
(!(z)[| z: A+B |] ==> [| C(z) type |]) ==>
[| ?a: (PROD x:A. C(inl(x))) --> (PROD y:B. C(inr(y)))
                  --> (PROD z: A+B. C(z)) |]
```

By the end of the proof, `?a` has become

```
lam ka. lam kb. lam kc. when(kc,%(kd)ka‘kd, %(kd)kb‘kd)
```

Type Theory satisfies a strong choice principle (Martin-Löf [10, page 50]). The proof requires a much more complicated series of commands than any of the others in this file.

```
[| A type |] ==>
(!(x)[| x:A |] ==> [| B(x) type |]) ==>
(!(x,y)[| x:A |] ==> [| y:B(x) |] ==> [| C(x,y) type |]) ==>
[| ?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))  |]
```

By the end of the proof, `?a` has become

```
lam ka. <lam u. fst(ka‘u), lam kb. snd(ka‘kb)>
```

The last two constructions are equivalent to those published by Martin-Löf [10].

Arithmetic

The largest example develops elementary arithmetic — the properties of addition, multiplication, subtraction, division, and remainder — culminating in the theorem

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

The declaration of `arith_thy`[4] demonstrates how to use `extend_theory`. The new theory extends the basic Type Theory syntax with six infix operators. Axioms define each operator in terms of others. Although here no operator is used before it is defined, Isabelle accepts arbitrary axioms without complaint. Compare with the definitions for set theory, below.

```
val arith_thy = Thm.extend_theory CTT_Rule.thy  "arith"
  [  (["#+","-","|-|"], InfixSy([Aexp,Aexp]--->Aexp, RightP 3)),
     (["#*","//","/"],  InfixSy([Aexp,Aexp]--->Aexp, RightP 5)) ]

  [ ("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//b == rec(a, 0, %(u,v)   \
  \                      rec(succ(v) |-| b, 0, %(x,y)succ(v)))"),
    ("quo_def",  "a/b == rec(a, 0, %(u,v)   \
  \                      rec(succ(u) // b, succ(v), %(x,y)v))") ]
```

Since Type Theory permits only primitive recursion, some of these definitions may be unfamiliar. The difference $a - b$ is computed by computing b predecessors of a; the predecessor function is

```
   %(v)rec(v, 0, %(x,y)x)
```

The remainder $a//b$ counts up to a in a cyclic fashion: whenever the count would reach b, the cyclic count returns to zero. Here the absolute difference gives an equality test. The quotient $a//b$ is computed by adding one for every number x such that $0 \leq x \leq a$ and $x//b = 0$.

The file proves many arithmetic laws, such as the following:

```
   [| a:N |] ==> [| b:N |] ==> [| a #+ b = b #+ a: N |]
```

[4]The theory is on `CTT/arith.ML`; sample proofs are on `CTT/ex/arith.ML`

`Aexp`	elements of types
`Atype`	types

Types

```
type  :  =  <  >  .  PROD  SUM  lam
```

Delimiters

symbol	*meta-type*	*precedence*	*description*
`	$[exp, exp] \to exp$	Left 7	function application
`*`	$[type, type] \to type$	Right 5	product of two types
`+`	$[type, type] \to type$	Right 4	sum of two types
`-->`	$[type, type] \to type$	Right 3	function type

Infixes

Figure 6: Lexical symbols for `CTT`

```
[| a:N |] ==> [| b:N |] ==> [| a #* b = b #* a: N  |]

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

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

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

Figure 7: The constants of `CTT`

Isabelle notation	*expansion*	*standard notation*
A `type`	`Type(`A`)`	the judgement A type
A `=` B	`Eqtype(`A`,`B`)`	the judgement $A = B$
a`:` A	`Elem(`a`,`A`)`	the judgement $a \in A$
a `=` b`:` A	`Eqelem(`a`,`b`,`A`)`	the judgement $a = b \in A$
`PROD` x`:` A`.` B	`Prod(`A`,` $\lambda x.B$`)`	the product $(\Pi x \in A)B$
`SUM` x`:` A`.` B	`Sum(`A`,` $\lambda x.B$`)`	the sum $(\Sigma x \in A)B$
`lam` x`.` b	`lambda(`$\lambda x.b$`)`	the abstraction $(\lambda x)b$
`<`a`,` b`>`	`pair(`a`,` b`)`	the pair $\langle a, b\rangle$

Notation

$$\begin{array}{rcl}
\textit{term} & = & \underline{\texttt{PROD}}\ \textit{var}\ \underline{\texttt{:}}\ \textit{type}\ \underline{\texttt{.}}\ \textit{type} \\
& | & \underline{\texttt{SUM}}\ \textit{var}\ \underline{\texttt{:}}\ \textit{type}\ \underline{\texttt{.}}\ \textit{type} \\
& | & \underline{\texttt{lam}}\ \textit{var}\ \underline{\texttt{.}}\ \textit{term} \\
& | & \underline{\texttt{<}}\ \textit{term}\ \underline{\texttt{,}}\ \textit{term}\ \underline{\texttt{>}} \\
& | & \underline{\texttt{[|}}\ \textit{judgement}\ \underline{\texttt{|]}} \\
& | & \textit{other Isabelle terms} \ldots
\end{array}$$

$$\begin{array}{rcl}
\textit{judgement} & = & \textit{type}\ \underline{\texttt{type}} \\
& | & \textit{type}\ \underline{\texttt{=}}\ \textit{type} \\
& | & \textit{term}\ \underline{\texttt{:}}\ \textit{type} \\
& | & \textit{term}\ \underline{\texttt{=}}\ \textit{term}\ \underline{\texttt{:}}\ \textit{type}
\end{array}$$

Syntactic definition

Figure 8: Syntax of `CTT`

```
Plus_form       [| A type |] ==> [| B type |] ==> [| A+B type |]
Plus_form_long  [| A = C |]  ==> [| B = D |]  ==> [| A+B = C+D |]
Plus_intr_inl   [| a: A |] ==> [| B type |] ==> [| inl(a): A+B |]
Plus_intr_inr   [| A type |] ==> [| b: B |] ==> [| inr(b): A+B |]
Plus_intr_inl_long
    [| a=c: A |] ==> [| B type |] ==> [| inl(a) = inl(c): A+B |]
Plus_intr_inr_long
    [| A type |] ==> [| b=d: B |] ==> [| inr(b) = inr(d): A+B |]
Plus_elim
    [| p: A+B |] ==>
    (!(x)[| x: A |] ==> [| c(x): C(inl(x)) |]) ==>
    (!(y)[| y: B |] ==> [| d(y): C(inr(y)) |]) ==>
    [| when(p,c,d): C(p) |]
Plus_elim_long
    [| 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,c,d) = when(q,e,f): C(p) |]
Plus_comp_inl
    [| a: A |] ==>
    (!(x)[| x: A |] ==> [| c(x): C(inl(x)) |]) ==>
    (!(y)[| y: B |] ==> [| d(y): C(inr(y)) |]) ==>
    [| when(inl(a),c,d) = c(a): C(inl(a)) |]
Plus_comp_inr
    [| b: B |] ==>
    (!(x)[| x: A |] ==> [| c(x): C(inl(x)) |]) ==>
    (!(y)[| y: B |] ==> [| d(y): C(inr(y)) |]) ==>
    [| when(inr(b),c,d) = d(b): C(inr(b)) |]
```

Figure 9: Meta-axioms for the + type

```
N_form           [| N type |]
N_intr0          [| 0: N |]
N_intr_succ      [| a: N |] ==> [| succ(a): N |]
N_intr_succ_long [| a = b: N |] ==> [| succ(a) = succ(b): N |]
N_elim
  [| p: N |] ==> [| a: C(0) |] ==>
  (!(u)[| u:N |] ==> !(v)[| v:C(u) |] ==> [| b(u,v):C(succ(u)) |])
  ==> [| rec(p,a,b): C(p) |]
N_elim_long
  [| p = q: N |] ==> [| a = c: C(0) |] ==>
  (!(u)[| u:N |] ==> !(v)[| v:C(u) |] ==>
            [| b(u,v)=d(u,v): C(succ(u)) |])
  ==> [| rec(p,a,b) = rec(q,c,d): C(p) |]
N_comp0
  [| a: C(0) |]  ==>
  (!(u)[| u:N |] ==> !(v)[| v:C(u) |] ==> [| b(u,v):C(succ(u)) |])
   ==> [| rec(0,a,b) = a: C(0) |]
N_comp_succ
  [| p: N |]  ==> [| a: C(0) |] ==>
  (!(u)[| u:N |] ==> !(v)[| v:C(u) |] ==> [| b(u,v):C(succ(u)) |])
  [| rec(succ(p),a,b) = b(p, rec(p,a,b)): C(succ(p)) |]
zero_ne_succ
  [| a: N |] ==> [| 0 = succ(a): N |] ==> [| 0: F |]
```

Figure 10: Meta-axioms for the type N

```
Prod_form
    [| A type |] ==> (!(w)[| w: A |] ==> [| B(w) type |]) ==>
    [| Prod(A,B) type |]
Prod_form_long
    [| A = C |] ==> (!(w)[| w: A |] ==> [| B(w) = D(w) |]) ==>
    [| Prod(A,B) = Prod(C,D) |]
Prod_intr
    [| A type |] ==>  (!(w)[| w: A |] ==> [| b(w): B(w) |]) ==>
    [| lambda(b): Prod(A,B) |]
Prod_intr_long
    [| A type |] ==> (!(w)[| w: A |] ==> [| b(w) = c(w): B(w) |])
    ==> [| lambda(b) = lambda(c): Prod(A,B) |]
Prod_elim
    [| p: Prod(A,B) |] ==> [| a: A |] ==> [| p ‘ a: B(a) |]
Prod_elim_long
    [| p = q: Prod(A,B) |] ==> [| a = b: A |] ==>
    [| p ‘ a = q ‘ b: B(a) |]
Prod_comp
    [| a: A |] ==> (!(w)[| w: A |] ==> [| b(w): B(w) |]) ==>
    [| lambda(b) ‘ a = b(a): B(a) |]
```

Figure 11: Meta-axioms for the product type

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

The judgement 'reduce'

```
Fun_def          A-->B == Prod(A, %(z)B)
Times_def        A*B == Sum(A, %(z)B)
fst_def          fst(a) == split(a, %(x,y)x)
snd_def          snd(a) == split(a, %(x,y)y)
```

Definitions

Figure 12: Other meta-axioms for CTT

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

Sum_elim_fst
    [| p: Sum(A,B) |] ==> [| fst(p): A |]

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

Figure 13: Derived rules for CTT

11 Classical first-order logic

The theory `LK` implements classical first-order logic through Gentzen's sequent calculus LK (see Gallier [6]). This calculus is especially suitable for purely backwards proof — it behaves almost exactly like the method of semantic tableaux. Assertions have the form $\Gamma \vdash \Delta$, where Γ and Δ are lists of formulae. Associative unification handles lists; we easily get powerful proof procedures.

11.1 Syntax and rules of inference

The module `LK_Syntax` defines the symbol table and syntax functions.

Figure 14 gives the lexical structure of `LK`. The types include formulae and expressions, and a type *sobj* used in the representation of lists. The actual list type, *sequ*, is just $sobj \to sobj$. The infixes are equality and the connectives.

Figure 15 gives the syntax for `LK`: sequents, quantifiers, descriptions, and negation. As in the intuitionistic theory `NJ` we may write `~P` instead of `~(P)`. Here negation is primitive and there is no constant `False`. Negation has a higher precedence than *all* infixes, so the parentheses in `~(a=b)` are necessary.

Traditionally Γ and Δ are sequence variables. Fixed variable declarations are inconvenient, so instead a dollar prefix designates sequence variables. In a sequence, any expression not prefixed by `$` is taken as a formula. For many examples of this notation, see below where the rules are presented.

The module `LK_Rule` declares meta-axioms (Figures 16 and 17) and binds them to ML identifiers of type `thm`. The connective $\leftrightarrow$ is defined using & and $\supset$.

Figure 18 presents derived rules, including rules for $\leftrightarrow$ and weakened quantifier rules. The automatic proof procedures, through these weakened rules, throw away each quantification after a single use. Thus they usually terminate quickly, but are incomplete. The multiple use of a quantifier can be obtained through a duplication rule. Recall that `lift_inst_tac` may be used to instantiate the variable `P` in these rules, specifying the formula to be duplicated.

11.2 Tactics for the cut rule

The theory `set`, which is built on `LK`, contains a good many derived rules. Many of the derivations use the cut rule. You might ask: what about cut-elimination? The cut rule can be eliminated from proofs of *sequents*, but it is still needed in derivations of *rules*.

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

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

The cut rule must be used, for $P \,\&\, Q$ is not a subformula of $Q \,\&\, P$.

A closer look at the derivations[5] shows that 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.

```
cut_tac: string -> int -> tactic
```

The tactic `cut_tac s i` reads the string s as an `LK` formula P, and applies the cut rule to subgoal i. The new subgoal i will have P on the right, while the new subgoal $i+1$ will have P on the left.

```
cut_right_tac: string -> int -> tactic
```

The tactic `cut_right_tac s i` is similar, but also deletes a formula from the right side of the new subgoal i. (It probably should delete all other formulae on the right.) Thus it is typically used to replace the right-hand formula by P.

```
cut_left_tac: string -> int -> tactic
```

The tactic `cut_left_tac s i` is similar, but also deletes a formula from the left side of the new subgoal $i+1$. Used to replace the left-hand formula by P.

11.3 Proof procedure

The `LK` proof procedure cannot compete with hand-coded theorem provers, but it is surprisingly powerful and natural. Because it is not restricted to a fixed set of rules, we may derive new rules and use them to derive others. Thus we can work directly with abstract concepts.

Rules are classified into *safe* and *unsafe*. An unsafe rule (typically a weakened quantifier rule) is only used when no safe rule can be. A *pack* is simply a pair whose first component is a list of safe rules, and whose second is a list of unsafe rules. Packs can be joined in an obvious way to allow reasoning with various fragments of the logic and its extensions.

For clarity, let us pretend that Isabelle declares the type `pack`. (Recall that type synonyms currently do not work with ML modules.)

```
type pack = thm list * thm list;
```

[5]for example on `LK/set/resolve.ML`

The module `LK_Resolve` makes the following declarations.

`triv_pack: pack`

This contains two trivial rules: reflexivity and the basic sequent.

`LK_pack: pack`

Its safe rules are all the propositional rules of `LK` plus `all_right` and `exists_left`. The unsafe rules are `all_left_thin` and `exists_right_thin`.

`pjoin: pack*pack -> pack`

Combines two packs into one in the obvious way: the lists of safe rules are concatenated, as are the lists of unsafe rules.

`filseq_resolve_tac: thm list -> int -> int -> tactic`

Calling `filseq_resolve_tac rules maxr i` determines which of the `rules` could affect a formula in subgoal `i`. If this number exceeds `maxr` then the tactic fails. Otherwise it behaves like `resolve_tac` (but runs much faster).

`reresolve_tac: thm list -> int -> tactic`

The tactic `reresolve_tac rules i` is like `rebires_tac` of Section 9.2. It repeatedly applies the given rules to subgoal i and the resulting subgoals.

`repeat_goal_tac: pack list -> int -> tactic`

This tactic implements the simple 'packs' idea. The safe rules in the packs are first applied as much as possible to a goal and resulting subgoals. At any such goal, if no safe rule is applicable then an unsafe rule is tried. For example, `disj_left` is tried before `all_left_thin`, even though `disj_left` would produce two subgoals.

`safe_goal_tac: pack list -> int -> tactic`

This tactic simply throws away the unsafe rules, calling `repeat_goal_tac` with the safe rules of the packs.

`step_tac: pack list -> int -> tactic`

For tracing a proof, `step_tac` applys just one rule. It tries the safe rules, then the unsafe rules.

`pc_tac: int -> tactic`

The basic predicate calculus prover, `pc_tac` attacks the numbered subgoal by calling `repeat_goal_tac` with `triv_pack` and `LK_pack`.

11.4 Examples

Several of Pelletier's problems [16] can be solved automatically. The following, numbers 20, 27, and 32, are of moderate difficulty.

```
[| H |- (ALL x. ALL y. EXISTS z. ALL w.(P(x)&Q(y)-->R(z)&S(w)))
       --> (EXISTS x. EXISTS y. P(x) & Q(y)) --> EXISTS z.R(z) |]

[|  EXISTS x. P(x) & ~Q(x),
    ALL x. P(x) --> R(x),
    ALL x. M(x) & L(x) --> P(x),
    (EXISTS x. R(x) & ~ Q(x)) --> (ALL x. L(x) --> ~ R(x))
    |- ALL x. M(x) --> ~L(x) |]

[|  ALL x. P(x) & (Q(x)|R(x))-->S(x),
    ALL x. S(x) & R(x) --> L(x),
    ALL x. M(x) --> R(x)
    |- ALL x. P(x) & M(x) --> L(x) |]
```

Backtracking over the choice of a safe rule (excluding the basic sequent) accomplishes nothing: applying them in any order leads to essentially the same result. This is intuitively clear enough, but a rigorous proof is essentially the Completeness Theorem [6].

Backtracking may be necessary over the choice of basic sequents. Suppose 0, 1, 2, 3 are constants in the subgoals

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

The only assignment that satisfies all three subgoals is $?a \mapsto 2$, and this can only be discovered by search.

`Aexp`	individuals
`Aform`	truth values
`Asobj`	dummy type for lists
`Asequ`	type of lists, $sequ = sobj \rightarrow sobj$

Types

```
$  |-  .  ~  ALL  EXISTS  THE
```

Delimiters

symbol	*meta-type*	*precedence*	*description*
`=`	$[exp, exp] \rightarrow form$	Left 6	equality ($=$)
`&`	$[form, form] \rightarrow form$	Right 5	conjunction (&)
`|`	$[form, form] \rightarrow form$	Right 4	disjunction ($\vee$)
`-->`	$[form, form] \rightarrow form$	Right 3	implication ($\supset$)
`<->`	$[form, form] \rightarrow form$	Right 3	if and only if ($\leftrightarrow$)

Infixes

symbol	*meta-type*	*description*
`True`	$sequ \rightarrow sequ \rightarrow prop$	meta-predicate of truth
`Seqof`	$form \rightarrow sequ$	singleton formula list
`Forall`	$(exp \rightarrow form) \rightarrow form$	universal quantifier ($\forall$)
`Exists`	$(exp \rightarrow form) \rightarrow form$	existential quantifier ($\exists$)
`The`	$(exp \rightarrow form) \rightarrow exp$	description operator (ϵ)
`not`	$form \rightarrow form$	negation ($\neg$)

Constants

Figure 14: Lexical symbols for `LK`

Isabelle notation	*expansion*	*standard notation*
Γ `|-` Δ	`True(`Γ`,` Δ`)`	sequent $\Gamma \vdash \Delta$
`$` Γ	Γ	sequence variable
`ALL` x`.` P	`Forall(`$\lambda x.P$`)`	universal quantification $\forall x.P$
`EXISTS` x`.` P	`Exists(`$\lambda x.P$`)`	existential quantification $\exists x.P$
`THE` x`.` P	`The(`$\lambda x.P$`)`	description $\epsilon x.P$
`~`P	`not(`P`)`	the negation $\neg P$

Notation

term	=	<u>`ALL`</u> *var* <u>`.`</u> *term*
	|	<u>`EXISTS`</u> *var* <u>`.`</u> *term*
	|	<u>`THE`</u> *var* <u>`.`</u> *term*
	|	<u>`~`</u> *term*
	|	<u>`[|`</u> *sequence* <u>`|-`</u> *sequence* <u>`|]`</u>
	|	*other Isabelle terms ...*
sequence	=	*item* { <u>`,`</u> *item* }
	|	*empty*
item	=	<u>`$`</u> *term*
	|	*term*

Syntactic definition

Figure 15: Syntax of `LK`

```
basic       [| $H, P, $G |- $E, P, $F |]
thin_right  [| $H |- $E, $F |] ==> [| $H |- $E, P, $F |]
thin_left   [| $H, $G |- $E |] ==> [| $H, P, $G |- $E |]
cut      [| $H |- $E, P |] ==> [| $H, P |- $E |] ==> [| $H |- $E |]
```

Structural rules

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

Equality rules

```
conj_right  [| $H |- $E, P, $F |] ==> [| $H |- $E, Q, $F |] ==>
            [| $H |- $E, P&Q, $F |]
conj_left   [| $H, P, Q, $G |- $E |] ==> [| $H, P&Q, $G |- $E |]

disj_right  [| $H |- $E, P, Q, $F |] ==> [| $H |- $E, P|Q, $F |]
disj_left   [| $H, P, $G |- $E |] ==> [| $H, Q, $G |- $E |] ==>
            [| $H, P|Q, $G |- $E |]

imp_right   [| $H, P |- $E, Q, $F |] ==> [| $H |- $E, P-->Q, $F |]
imp_left    [| $H,$G |- $E,P |] ==> [| $H, Q, $G |- $E |] ==>
            [| $H, P-->Q, $G |- $E |]

not_right   [| $H, P |- $E, $F |] ==> [| $H |- $E, ~P, $F |]
not_left    [| $H, $G |- $E, P |] ==> [| $H, ~P, $G |- $E |]

iff_def     P<->Q == (P-->Q) & (Q-->P)
```

Propositional rules

Figure 16: Meta-axioms for LK

```
all_right   (!(x)[| $H |- $E, P(x), $F |]) ==>
            [| $H |- $E, Forall(P), $F |]
all_left    [| $H, P(a), $G, Forall(P) |- $E |] ==>
            [| $H, Forall(P), $G |- $E |]

exists_right [| $H |- $E, P(a), $F, Exists(P) |] ==>
             [| $H |- $E, Exists(P), $F |]
exists_left (!(x)[| $H, P(x), $G |- $E |]) ==>
            [| $H, Exists(P), $G |- $E |]

The         [| $H |- $E, P(a), $F |] ==>
            (!(y)[| $H, P(y) |- $E, y=a, $F |]) ==>
            [| $H |- $E, P(The(P)), $F |]
```

Figure 17: Quantifier and description rules

```
duplicate_right [| $H |- $E, P, $F, P |] ==> [| $H |- $E, P, $F |]
duplicate_left  [| $H, P, $G, P |- $E |] ==> [| $H, P, $G |- $E |]

iff_right   [| $H,P |- $E,Q,$F |] ==> [| $H,Q |- $E,P,$F |] ==>
            [| $H |- $E, P <-> Q, $F |]
iff_left    [| $H,$G |- $E,P,Q |] ==> [| $H,Q,P,$G |- $E |] ==>
            [| $H, P <-> Q, $G |- $E |]
```

Duplication and 'if and only if'

```
all_left_thin
    [| $H, P(a), $G |- $E |] ==> [| $H, Forall(P), $G |- $E |]
exists_right_thin
    [| $H |- $E, P(a), $F |] ==> [| $H |- $E, Exists(P), $F |]
```

Weakened quantifier rules

Figure 18: Derived rules for LK

12 Zermelo-Fraenkel set theory

The Isabelle theory called `set` implements Zermelo-Fraenkel set theory [19]. It rests on classical first-order logic, `LK`. Isabelle expresses the notorious axiom schemes (selection and replacement) using function variables.

The theory includes a collection of derived rules that form a sequent calculus of sets. The simplistic sequent calculus proof procedure that was developed for `LK` works reasonably well for set theory.

12.1 Syntax and rules of inference

The module `Set_Syntax` defines the symbol table. The parsing and printing functions handle set theory notation and otherwise call the syntax functions for classical logic. Figure 19 gives the lexical structure of `set`. There are no types beyond those of `LK`. Infixes include union and intersection, and the subset and membership relations. Besides 2-place union and intersection ($A \cup B$ and $A \cap B$) we have 'big union' and 'big intersection' operators ($\bigcup C$ and $\bigcap C$). These form the union or intersection of a set of sets;[6] $\bigcup C$ can also be written $\bigcup_{A \in C} A$. Of these operators only 'big union' is primitive.

The standard language of ZF set theory has no constants. The empty set axiom asserts that some set is empty, not that $\emptyset$ is the empty set; and so on for union, powerset, etc. Formal proofs in this language would be barbarous. The Isabelle theory declares primitive and defined constants.

Figure 20 gives the syntax for `set`, which extends `LK` with finite sets, ordered pairs, and comprehension. The constant :: is a 'set cons', for $a :: B = \{a\} \cup B$. It constructs finite sets in the obvious way:

$$\{a, b, c, d\} \quad = \quad a :: (b :: (c :: (d :: \emptyset)))$$

ZF set theory permits limited comprehension. By the separation axiom, the set `Collect(`A`,`P`)` forms the set of all $x \in A$ that satisfy $P(x)$. By the replacement axiom, the set `Replace(`f`,`A`)` forms the set of all $f(x)$ for $x \in A$. The syntax of `set` can express three kinds of comprehension: separation, replacement, and both together.

The module `Set_Rule` binds the axioms of set theory to ML identifiers. These axioms appear in Figures 21 and 22. They contain unusual definitions where one *formula* is defined to denote another. The extensionality axiom states that $A = B$ means the same thing as $A \subseteq B \,\&\, B \subseteq A$. The power set axiom states that $A \in$

[6]In the latest jargon, set theory has 'dependent sets'.

Pow(B) means the same thing as $A \subseteq B$. Such definitions need not be conservative since they are not simply abbreviations. The theory also defines the traditional abbreviations for ordered pairs, successor, etc.

The ZF axioms can be expressed in many different ways. For example, the axiom `equal_members` could be expressed as

$$\forall xyA\,.\,(x = y \,\&\, y \in A) \supset x \in A$$

But applying this axiom would require using several `LK` rules. (Most books on set theory omit this axiom altogether!) The axiom of regularity is expressed in its most useful form: transfinite induction.

The replacement axiom involves the concept of *class function*, which is like a function defined on the entire universe of sets. Examples include the power set operator and the successor operator $\mathrm{succ}(x) = x \cup \{x\}$. In set theory, a function *is* its graph. Since the graph of a class function is 'too big' to be a set, it is represented by a 2-place predicate. The theory `set` assumes that every class function can be expressed by some Isabelle term — possibly involving `LK`'s description operator ('The').

12.2 Derived rules

The module `LK_Resolve` derives a sequent calculus from the set theory axioms. Figures 23–25 present most of the rules, which refer to the constants of `set` rather than the logical constants.

A rule named X`_thin` has been weakened. In a typical weakened rule:

- A formula in the conclusion is omitted in the premises to allow repeated application of the rule without looping — but this proof procedure is incomplete.
- Some variables (`Vars`) appear in the premises only, not in the conclusion. In backwards proof these rules introduce new variables in the subgoals.

Recall that a rule is called *unsafe* if it can reduce a provable goal to unprovable subgoals. The rule `subset_left_thin` uses the fact $A \subseteq B$ to reason, 'for any c, if $c \in A$ then $c \in B$.' It reduces $A \subseteq B \vdash A \subseteq B$, which is obviously valid, to the subgoals $\vdash A \subseteq B, ?c \in A$ and $?c \in B \vdash A \subseteq B$. These are not valid: if $A = \{2\}$, $B = \{1\}$, and $?c = 1$ then both subgoals are false.

A safe variant of the rule would reduce $A \subseteq B \vdash A \subseteq B$ to the subgoals $A \subseteq B \vdash A \subseteq B, c \in A$ and $A \subseteq B, c \in B \vdash A \subseteq B$, both trivially valid. In contrast, `subset_right` is safe: if the conclusion is true, then $A \subseteq B$, and thus the premise is also true: if $x \in A$ then $x \in B$ for arbitrary x.

The rules for big intersection are not completely analogous to those for big union. Consider applying these operators to the empty set. Clearly $\bigcup(\emptyset)$ equals $\emptyset$, as it should. We might expect $\bigcap(\emptyset)$ to equal the universal set, but there is no such thing in ZF set theory. The definition perversely makes $\bigcap(\emptyset)$ equal $\emptyset$; we may as well regard it as undefined. The rule `Inter_right` lets us prove $A \in \bigcap(C)$ by proving that $x \in C$ implies $A \in x$ for every x, but a second subgoal requires us to consider that C could be empty.

Another collection of derived rules considers the set operators under the subset relation, as in $A \cup B \subseteq C$. These are not shown here.

12.3 Tactics

The `set` theorem prover uses the 'pack' techniques of `LK`. The set theory sequent calculus lets us prove many theorems about sets without ever seeing a logical connective. Such proofs are more direct and efficient because they do not involve the rules of `LK` [18]. Putting packs together gives various combinations of rules to `repeat_goal_tac`. This should illustrate the purpose of the pack mechanism.

Equality reasoning is difficult at present. While the extensionality rules can do a surprising amount with equalities, we need a simplifier. Subgoal reordering sometimes appears necessary; some methods that work in `CTT` might be adopted.

```
set_pack: pack
```

Holds the sequent rules for set theory.

```
ext_pack: pack
```

The extensionality rules (which treat $A = B$ like $A \subseteq B \,\&\, B \subseteq A$) are not included in `set_pack` because they can be expensive to use. The rules are `equal_right` (safe) and `eqext_left_thin` (unsafe).

```
set_tac: int -> tactic
```

The set theory prover works by calling `repeat_goal_tac` with `triv_pack` and `set_pack`.

```
setpc_tac: int -> tactic
```

This uses both set theory and predicate calculus rules, calling `repeat_goal_tac` with `triv_pack`, `set_pack`, and `LK_pack`.

```
set_step_tac: int -> tactic
```

For debugging, does a single step in set theory by calling `step_tac` with `triv_pack` and `set_pack`.

```
    setpc_step_tac: int -> tactic
```

Applies one set theory or predicate calculus rule, calling `step_tac` with `triv_pack`, `set_pack`, and `LK_pack`.

12.4 Examples

The Isabelle distribution includes several examples files for set theory.[7]

A simple example about unions and intersections is

```
[| |- C<=A  <->  (A Int B) Un C = A Int (B Un C) |]
```

Proofs about 'big intersection' tend to be complicated because $\bigcap$ is ill-behaved on the empty set. Two interesting examples are

```
[|  |- Inter(A Un B) = Inter(A) Int Inter(B), A<=0, B<=0 |]

[| ~(C<=0) |- Inter([ A(x) Int B(x) || x:C ]) =
       Inter([ A(x) || x:C ])  Int  Inter([ B(x) || x:C ]) |]
```

In traditional notation these are

$$A \neq \emptyset \,\&\, B \neq \emptyset \supset \bigcap(A \cup B) = (\bigcap A) \cap (\bigcap B)$$

$$C \neq \emptyset \supset \bigcap_{x \in C} (A(x) \cap B(x)) = (\bigcap_{x \in C} A(x)) \cap (\bigcap_{x \in C} B(x))$$

Observe how replacement is used to construct families for intersection.

Another large example justifies the standard definition of pairing:

$$\langle a, b \rangle \quad = \quad \{\{a\}, \{a, b\}\}$$

It proves that $\langle a, b \rangle = \langle c, d \rangle$ implies $a = c$ and $b = d$. Try proving this yourself from the axioms.

[7]the files `un-int.ML`, `big-un-int.ML`, `power.ML`, `pairing.ML`, and `prod.ML` on directory `LK/set/ex`

`{  }  <  >  [  ]  ||`

Delimiters

symbol	*meta-type*	*precedence*	*description*
`	$[exp, exp] \rightarrow exp$	Left 9	function application
`Int`	$[exp, exp] \rightarrow exp$	Right 8	intersection ($\cap$)
`Un`	$[exp, exp] \rightarrow exp$	Right 7	union ($\cup$)
`-`	$[exp, exp] \rightarrow exp$	Right 7	difference ($-$)
`::`	$[exp, exp] \rightarrow exp$	Right 7	inclusion of an element
`<=`	$[exp, exp] \rightarrow form$	Right 6	subset ($\subseteq$)
`:`	$[exp, exp] \rightarrow form$	Right 6	membership ($\in$)

Infixes

symbol	*meta-type*	*description*
`0`	exp	empty set
`INF`	exp	infinite set
`Pow`	$exp \rightarrow exp$	powerset operator
`Union`	$exp \rightarrow exp$	'big union' operator
`Inter`	$exp \rightarrow exp$	'big intersection' operator
`Pair`	$[exp, exp] \rightarrow exp$	pairing operator
`succ`	$exp \rightarrow exp$	successor operator
`Choose`	$exp \rightarrow exp$	choice operator
`Collect`	$[exp, exp \rightarrow form] \rightarrow exp$	separation operator
`Replace`	$[exp \rightarrow exp, exp] \rightarrow exp$	replacement operator

Constants

Figure 19: Lexical symbols for `set`

Isabelle notation	*expansion*	*standard notation*
`{` a_1`,` ...`,` a_n `}`	a_1`::`$\cdots$`::(`a_n`::0)`	$\{a_1, \ldots, a_n\}$
`<` a`,` b `>`	`Pair(`a`,`b`)`	$\langle a, b\rangle$
`[` x `||` x`:`A`,` $P(x)$ `]`	`Collect(`A`,`P`)`	$\{x \in A \mid P[x]\}$
`[` $f(x)$ `||` x`:`A `]`	`Replace(`f`,`A`)`	$\{f[x] \mid x \in A\}$
`[` $f(x)$ `||` x`:`A`,`$P(x)$ `]`	`Replace(`f`,Collect(`A`,`P`))`	$\{f[x] \mid x \in A \,\&\, P[x]\}$

NOTATION

$$
\begin{array}{rcl}
\textit{term} & = & \underline{\texttt{\{}}\ \textit{term}\ \{\ \underline{\texttt{,}}\ \textit{term}\ \}\ \underline{\texttt{\}}} \\
& | & \underline{\texttt{<}}\ \textit{term}\ \underline{\texttt{,}}\ \textit{term}\ \underline{\texttt{>}} \\
& | & \underline{\texttt{[}}\ \textit{var}\ \underline{\texttt{||}}\ \textit{var}\ \underline{\texttt{:}}\ \textit{term}\ \underline{\texttt{,}}\ \textit{term}\ \underline{\texttt{]}} \\
& | & \underline{\texttt{[}}\ \textit{term}\ \underline{\texttt{||}}\ \textit{var}\ \underline{\texttt{:}}\ \textit{term}\ \underline{\texttt{]}} \\
& | & \underline{\texttt{[}}\ \textit{term}\ \underline{\texttt{||}}\ \textit{var}\ \underline{\texttt{:}}\ \textit{term}\ \underline{\texttt{,}}\ \textit{form}\ \underline{\texttt{]}} \\
& | & \textit{other}\ \texttt{LK}\ \textit{terms}\ \ldots
\end{array}
$$

SYNTACTIC DEFINITION

Figure 20: Syntax of `set`

```
null_left       [| $H, a: 0, $G |- $E |]
setcons_def     a: (b::B)  ==  a=b | a:B
Pair_def        <a,b>  ==  { {a}, {a,b} }
```

EMPTY SET, FINITE SETS, AND ORDERED PAIRS

```
subset_def      A<=B  ==  ALL x. x:A --> x:B
equal_members   [| $H |- $E, a=b, $F |] ==> [| $H |- $E, b:A, $F |]
            ==> [| $H |- $E, a:A, $F |]
ext_def         A=B   ==  A<=B & B<=A
```

SUBSETS, EQUALITY, EXTENSIONALITY

Figure 21: Meta-axioms for `set`

```
Pow_def         A: Pow(B)  ==  A<=B
Collect_def     a: Collect(A,P)  ==  a:A & P(a)
Replace_def     c: Replace(f,B)  == EXISTS a. a:B & c=f(a)
```

Power set, Separation, Replacement

```
Union_def      A: Union(C)  ==  EXISTS B. A:B  &  B:C
Un_def              a Un b  ==  Union({a,b})
```

Union

```
Inter_def    Inter(C)  ==  [ x || x: Union(C), ALL y. y:C --> x:y ]
Int_def       a Int b  ==  [ x || x:a, x:b ]
Diff_def          a-b  ==  [ x || x:a, ~(x:b) ]
```

Intersection and Difference

```
succ_def            succ(a)  ==  a Un {a}
INF_right_0         [| $H |- $E, 0:INF, $F |]
INF_right_succ      [| $H |- $E, a:INF --> succ(a):INF, $F |]
```

Infinity

```
Choose  [| $H, A=0 |- $E, $F |] ==> [| $H |- $E, Choose(A):A, $F |]

induction    (!(u) [| $H, ALL v. v:u --> P(v) |- $E, P(u), $F |])
         ==> [| $H |- $E, P(a), $F |]
```

Choice and Transfinite Induction

Figure 22: Meta-axioms for set (continued)

```
null_right      [| $H |- $E, $F |] ==> [| $H |- $E, a:0, $F |]

setcons_right   [| $H |- $E, a=b, a:B, $F |] ==>
                [| $H |- $E, a: (b::B), $F |]
setcons_left
    [| $H, a=b, $G |- $E |] ==> [| $H, a:B, $G |- $E |] ==>
    [| $H, a:(b::B), $G |- $E |]
```

Empty and Finite Sets

```
subset_right    (!(x)[| $H, x:A |- $E, x:B, $F |]) ==>
                [| $H |- $E, A <= B, $F |]
subset_left_thin
    [| $H, $G |- $E, c:A |] ==> [| $H, c:B, $G |- $E |] ==>
    [| $H, A <= B, $G |- $E |]

equal_right
    [| $H |- $E, A<=B, $F |] ==> [| $H |- $E, B<=A, $F |] ==>
    [| $H |- $E, A=B, $F |]
equal_left_s    [| $H, A<=B, B<=A, $G |- $E |] ==>
                [| $H, A=B, $G |- $E |]
eqext_left_thin
  [| $H, $G |- $E, c:A, c:B |] ==> [| $H, c:B, c:A, $G |- $E |] ==>
  [| $H, A=B, $G |- $E |]
eqmem_left_thin
  [| $H, $G |- $E, a:c, b:c |] ==> [| $H, $G, b:c, a:c |- $E |] ==>
  [| $H, a=b, $G |- $E |]
```

Subsets, Equality, Extensionality

Figure 23: The derived sequent calculus for `set`

```
Union_right_thin
    [| $H |- $E, A:B, $F |] ==> [| $H |- $E, B:C, $F |] ==>
    [| $H |- $E, A: Union(C), $F |]
Union_left      (!(x)[| $H, A:x, x:C, $G |- $E |]) ==>
                [| $H, A:Union(C), $G |- $E |]

Un_right        [| $H |- $E, $F, c:A, c:B |] ==>
                [| $H |- $E, c: A Un B, $F |]
Un_left [| $H, c:A, $G |- $E |] ==> [| $H, c:B, $G |- $E |] ==>
        [| $H, c: A Un B, $G |- $E |]
```

Union

```
Inter_right
  (!(x)[| $H, x:C |- $E, A:x, $F |]) ==> [| $H, C<=0 |- $E, $F |]
     ==> [| $H |- $E, A: Inter(C), $F |]
Inter_left_thin
    [| $H, A:B, $G |- $E |] ==> [| $H, $G |- $E, B:C |] ==>
    [| $H, A: Inter(C), $G |- $E |]

Int_right [| $H |- $E, c:A, $F |] ==> [| $H |- $E, c:B, $F |] ==>
          [| $H |- $E, c: A Int B, $F |]
Int_left        [| $H, c:A, c:B, $G |- $E |] ==>
                [| $H, c: A Int B, $G |- $E |]

Diff_right  [| $H |- $E, c:A, $F |] ==> [| $H, c:B |- $E, $F |] ==>
            [| $H |- $E, c:A-B, $F |]
Diff_left       [| $H, c:A, $G |- $E, c:B |] ==>
                [| $H, c: A-B, $G |- $E |]
```

Intersection and Difference

Figure 24: The derived sequent calculus for `set` (continued)

```
Pow_right [| $H |- $E, A<=B, $F |] ==> [| $H |- $E, A:Pow(B), $F |]
Pow_left  [| $H, A<=B, $G |- $E |] ==> [| $H, A:Pow(B), $G |- $E |]

Collect_right
    [| $H |- $E, a:A, $F |] ==> [| $H |- $E, P(a), $F |] ==>
    [| $H |- $E, a: Collect(A,P), $F |]
Collect_left    [| $H, a: A, P(a), $G |- $E |] ==>
                [| $H, a: Collect(A,P), $G |- $E |]

Replace_right_comb   [| $H |- $E, a:B, $F |] ==>
                     [| $H |- $E, f(a):Replace(f,B), $F |]
Replace_right_thin
  [| $H |- $E, a:B, $F |] ==> [| $H |- $E, c=f(a), $F |] ==>
  [| $H |- $E, c: Replace(f,B), $F |]
Replace_left    (!(x)[| $H, x:B, c=f(x), $G |- $E |]) ==>
                [| $H, c: Replace(f,B), $G |- $E |]
```

Power set, Separation, Replacement

Figure 25: The derived sequent calculus for `set` (continued)

References

[1] R. S. Boyer, J S. Moore, The sharing of structure in theorem-proving programs, in: B. Meltzer and D. Michie, editors, *Machine Intelligence 7* (Edinburgh University Press, 1972), pages 101–116.

[2] N. G. de Bruijn, Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem, *Indagationes Mathematicae* **34** (1972), pages 381–392.

[3] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panagaden, J. T. Sasaki, S. F. Smith, *Implementing Mathematics with the Nuprl Proof Development System* (Prentice-Hall, 1986).

[4] Th. Coquand, An analysis of Girard's paradox, *Symposium on Logic in Computer Science* (IEEE Computer Society Press, 1986), pages 227–236.

[5] M. Dummett, *Elements of Intuitionism* (Oxford University Press, 1977).

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

[7] Ph. de Groote, How I spent my time in Cambridge with Isabelle, Report RR 87-1, Unité d'informatique, Université Catholique de Louvain, Belgium (1987).

[8] G. P. Huet, A unification algorithm for typed λ-calculus, *Theoretical Computer Science* **1** (1975), pages 27–57.

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

[10] P. Martin-Löf, *Intuitionistic type theory* (Bibliopolis, 1984).

[11] P. Martin-Löf, Amendment to intuitionistic type theory, Lecture notes obtained from P. Dybjer, Computer Science Department, Chalmers University, Gothenburg (1986).

[12] G. Nadathur, *A Higher-Order Logic as the Basis for Logic Programming*, PhD Thesis, University of Pennsylvania (1987).

[13] L. C. Paulson, Natural deduction as higher-order resolution, *Journal of Logic Programming* **3** (1986), pages 237–258.

[14] L. C. Paulson, *Logic and Computation: Interactive Proof with Cambridge* (Cambridge University Press, 1987).

[15] L. C. Paulson, The foundation of a generic theorem prover, Report 130, Computer Lab., Univ. of Cambridge (1987).

[16] F. J. Pelletier, Seventy-five problems for testing automatic theorem provers, *Journal of Automated Reasoning* **2** (1986), pages 191–216.

[17] D. Sannella, R. M. Burstall, Structured theories in LCF, in: G. Ausiello, M. Protasi (editors), *Eighth Colloquium on Trees in Algebra and Programming* (Springer, 1983), pages 377–391.

[18] D. Schmidt, Natural deduction theorem proving in set theory, Report CSR-142-83, Dept. of Computer Science, Univ. of Edinburgh (1983).

[19] P. Suppes, *Axiomatic Set Theory* (Dover Publications, 1972).

[20] L. A. Wallen, *Automated Proof Search in Non-classical Logics: Efficient Matrix Proof Methods for Modal and Intuitionistic Logics*, PhD Thesis, University of Edinburgh (1987).

[21] Å. Wikström, *Functional Programming Using Standard ML* (Prentice-Hall, 1987).