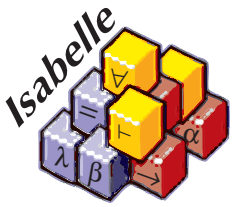

Isabelle's Logics: HOL[1]

Tobias Nipkow[2] and Lawrence C. Paulson[3] and Markus Wenzel[4]

12 February 2013

[1]The research has been funded by the EPSRC (grants GR/G53279, GR/H40570, GR/K57381, GR/K77051, GR/M75440), by ESPRIT (projects 3245: Logical Frameworks, and 6453: Types) and by the DFG Schwerpunktprogramm *Deduktion*.

[2]Institut für Informatik, Technische Universität München, `nipkow@in.tum.de`

[3]Computer Laboratory, University of Cambridge, `lcp@cl.cam.ac.uk`

[4]Institut für Informatik, Technische Universität München, `wenzelm@in.tum.de`

Abstract

This manual describes Isabelle's formalization of Higher-Order Logic, a polymorphic version of Church's Simple Theory of Types. HOL can be best understood as a simply-typed version of classical set theory. The monograph *Isabelle/HOL — A Proof Assistant for Higher-Order Logic* provides a gentle introduction on using Isabelle/HOL in practice.

Contents

Chapter 1

Syntax definitions

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

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

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

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

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

variable. ZF's bounded quantifier $\forall x \in A \,.\, P(x)$ cannot be declared as a binder because it has type $[i, i \Rightarrow o] \Rightarrow o$. The syntax for binders allows type constraints on bound variables, as in

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

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

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

Chapter 2

Higher-Order Logic

2.1 Syntax

Figure 2.1 lists the constants (including infixes and binders), while Fig. 2.2 presents the grammar of higher-order logic. Note that a~=b is translated to $\neg(a = b)$.

! HOL has no if-and-only-if connective; logical equivalence is expressed using equality. But equality has a high priority, as befitting a relation, while if-and-only-if typically has the lowest priority. Thus, $\neg\neg P = P$ abbreviates $\neg\neg(P = P)$ and not $(\neg\neg P) = P$. When using $=$ to mean logical equivalence, enclose both operands in parentheses.

2.1.1 Types and overloading

The universal type class of higher-order terms is called `term`. By default, explicit type variables have class `term`. In particular the equality symbol and quantifiers are polymorphic over class `term`.

The type of formulae, *bool*, belongs to class `term`; thus, formulae are terms. The built-in type *fun*, which constructs function types, is overloaded with arity `(term, term) term`. Thus, $\sigma \Rightarrow \tau$ belongs to class `term` if σ and τ do, allowing quantification over functions.

HOL allows new types to be declared as subsets of existing types, either using the primitive `typedef` or the more convenient `datatype` (see §2.6).

Several syntactic type classes — `plus`, `minus`, `times` and `power` — permit overloading of the operators `+`, `-`, `*`. and `^`. They are overloaded to denote the obvious arithmetic operations on types `nat`, `int` and `real`. (With the `^` operator, the exponent always has type `nat`.) Non-arithmetic overloadings are also done: the operator `-` can denote set difference, while `^` can denote exponentiation of relations (iterated composition). Unary minus is also written as `-` and is overloaded like its 2-place counterpart; it even can stand for set complement.

The constant `0` is also overloaded. It serves as the zero element of several types, of which the most important is `nat` (the natural numbers). The type

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

CONSTANTS

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

BINDERS

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

INFIXES

Figure 2.1: Syntax of `HOL`

term	=	expression of class *term*
	|	`SOME` *id* `.` *formula* | `@` *id* `.` *formula*
	|	`let` *id* `=` *term*`;` ... `;` *id* `=` *term* `in` *term*
	|	`if` *formula* `then` *term* `else` *term*
	|	`LEAST` *id* `.` *formula*
formula	=	expression of type *bool*
	|	*term* `=` *term*
	|	*term* `~=` *term*
	|	*term* `<` *term*
	|	*term* `<=` *term*
	|	`~` *formula*
	|	*formula* `&` *formula*
	|	*formula* `|` *formula*
	|	*formula* `-->` *formula*
	|	`ALL` *id* *id** `.` *formula* | `!` *id* *id** `.` *formula*
	|	`EX` *id* *id** `.` *formula* | `?` *id* *id** `.` *formula*
	|	`EX!` *id* *id** `.` *formula* | `?!` *id* *id** `.` *formula*

Figure 2.2: Full grammar for HOL

class `plus_ac0` comprises all types for which 0 and + satisfy the laws $x+y = y+x$, $(x+y)+z = x+(y+z)$ and $0+x = x$. These types include the numeric ones `nat`, `int` and `real` and also multisets. The summation operator `setsum` is available for all types in this class.

Theory `Ord` defines the syntactic class `ord` of order signatures. The relations $<$ and $\leq$ are polymorphic over this class, as are the functions `mono`, `min` and `max`, and the `LEAST` operator. `Ord` also defines a subclass `order` of `ord` which axiomatizes the types that are partially ordered with respect to $\leq$. A further subclass `linorder` of `order` axiomatizes linear orderings. For details, see the file `Ord.thy`.

If you state a goal containing overloaded functions, you may need to include type constraints. Type inference may otherwise make the goal more polymorphic than you intended, with confusing results. For example, the variables i, j and k in the goal $i \leq j \Longrightarrow i \leq j+k$ have type $\alpha :: \{ord, plus\}$, although you may have expected them to have some numeric type, e.g. *nat*. Instead you should have stated the goal as $(i :: nat) \leq j \Longrightarrow i \leq j+k$, which causes all three variables to have type *nat*.

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

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

2.1.2 Binders

Hilbert's **description** operator $\varepsilon x \,.\, P[x]$ stands for some x satisfying P, if such exists. Since all terms in HOL denote something, a description is always meaningful, but we do not know its value unless P defines it uniquely. We may write descriptions as `Eps`$(\lambda x \,.\, P[x])$ or use the syntax `SOME` x`.` $P[x]$.

Existential quantification is defined by

$$\exists x \,.\, P\ x \ \equiv\ P(\varepsilon x \,.\, P\ x).$$

The unique existence quantifier, $\exists!x \,.\, P$, is defined in terms of $\exists$ and $\forall$. An Isabelle binder, it admits nested quantifications. For instance, $\exists!x\ y \,.\, P\ x\ y$ abbreviates $\exists!x \,.\, \exists!y \,.\, P\ x\ y$; note that this does not mean that there exists a unique pair (x, y) satisfying $P\ x\ y$.

The basic Isabelle/HOL binders have two notations. Apart from the usual `ALL` and `EX` for $\forall$ and $\exists$, Isabelle/HOL also supports the original notation of Gordon's HOL system: `!` and `?`. In the latter case, the existential quantifier *must* be followed by a space; thus `?x` is an unknown, while `? x. f x=y` is a quantification. Both notations are accepted for input. The print mode "`HOL`" governs the output notation. If enabled (e.g. by passing option `-m HOL` to the `isabelle` executable), then `!` and `?` are displayed.

If τ is a type of class `ord`, P a formula and x a variable of type τ, then the term `LEAST` x . $P[x]$ is defined to be the least (w.r.t. $\leq$) x such that $P\ x$ holds (see Fig. 2.4). The definition uses Hilbert's ε choice operator, so `Least` is always meaningful, but may yield nothing useful in case there is not a unique least element satisfying P.[1]

All these binders have priority 10.

! The low priority of binders means that they need to be enclosed in parenthesis when they occur in the context of other operations. For example, instead of $P \wedge \forall x\ .\ Q$ you need to write $P \wedge (\forall x\ .\ Q)$.

2.1.3 The let and case constructions

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

HOL also defines the basic syntax

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

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

! Both `if` and `case` constructs have as low a priority as quantifiers, which requires additional enclosing parentheses in the context of most other operations. For example, instead of $f\ x = \texttt{if}\ldots\texttt{then}\ldots\texttt{else}\ldots$ you need to write $f\ x = (\texttt{if}\ldots\texttt{then}\ldots\texttt{else}\ldots)$.

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

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

Figure 2.3: The `HOL` rules

2.2 Rules of inference

Figure 2.3 shows the primitive inference rules of HOL, with their ML names. Some of the rules deserve additional comments:

`ext` expresses extensionality of functions.

`iff` asserts that logically equivalent formulae are equal.

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

`True_or_False` makes the logic classical.[2]

HOL follows standard practice in higher-order logic: only a few connectives are taken as primitive, with the remainder defined obscurely (Fig. 2.4). Gordon's HOL system expresses the corresponding definitions [2, page 270] using object-equality (`=`), which is possible because equality in higher-order logic may equate formulae and even functions over formulae. But theory HOL, like all other Isabelle theories, uses meta-equality (`==`) for definitions.

! The definitions above should never be expanded and are shown for completeness only. Instead users should reason in terms of the derived rules shown below or, better still, using high-level tactics.

Some of the rules mention type variables; for example, `refl` mentions the type variable `'a`. This allows you to instantiate type variables explicitly by calling `res_inst_tac`.

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

```
True_def   True     == ((%x::bool. x)=(%x. x))
All_def    All      == (%P. P = (%x. True))
Ex_def     Ex       == (%P. P(@x. P x))
False_def  False    == (!P. P)
not_def    not      == (%P. P-->False)
and_def    op &     == (%P Q. !R. (P-->Q-->R) --> R)
or_def     op |     == (%P Q. !R. (P-->R) --> (Q-->R) --> R)
Ex1_def    Ex1      == (%P. ? x. P x & (! y. P y --> y=x))

o_def      op o     == (%(f::'b=>'c) g x::'a. f(g x))
if_def     If P x y ==
              (%P x y. @z::'a.(P=True --> z=x) & (P=False --> z=y))
Let_def    Let s f  == f s
Least_def  Least P  == @x. P(x) & (ALL y. P(y) --> x <= y)"
```

Figure 2.4: The `HOL` definitions

Some derived rules are shown in Figures 2.5 and 2.6, with their ML names. These include natural rules for the logical connectives, as well as sequent-style elimination rules for conjunctions, implications, and universal quantifiers.

Note the equality rules: `ssubst` performs substitution in backward proofs, while `box_equals` supports reasoning by simplifying both sides of an equation.

The following simple tactics are occasionally useful:

`strip_tac` i applies `allI` and `impI` repeatedly to remove all outermost universal quantifiers and implications from subgoal i.

`case_tac "`P`"` i performs case distinction on P for subgoal i: the latter is replaced by two identical subgoals with the added assumptions P and $\neg P$, respectively.

`smp_tac` j i applies j times `spec` and then `mp` in subgoal i, which is typically useful when forward-chaining from an induction hypothesis. As a generalization of `mp_tac`, if there are assumptions $\forall \vec{x} \,.\, P\vec{x} \to Q\vec{x}$ and $P\vec{a}$, ($\vec{x}$ being a vector of j variables) then it replaces the universally quantified implication by $Q\vec{a}$. It may instantiate unknowns. It fails if it can do nothing.

2.3 A formulation of set theory

Historically, higher-order logic gives a foundation for Russell and Whitehead's theory of classes. Let us use modern terminology and call them **sets**, but

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

EQUALITY

```
TrueI       True
FalseE      False ==> P

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

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

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

PROPOSITIONAL LOGIC

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

LOGICAL EQUIVALENCE

Figure 2.5: Derived rules for HOL

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

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

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

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

Quantifiers and descriptions

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

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

Classical logic

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

Conditionals

Figure 2.6: More derived rules

name	*meta-type*	*description*
`{}`	$\alpha\ set$	the empty set
`insert`	$[\alpha, \alpha\ set] \Rightarrow \alpha\ set$	insertion of element
`Collect`	$(\alpha \Rightarrow bool) \Rightarrow \alpha\ set$	comprehension
`INTER`	$[\alpha\ set, \alpha \Rightarrow \beta\ set] \Rightarrow \beta\ set$	intersection over a set
`UNION`	$[\alpha\ set, \alpha \Rightarrow \beta\ set] \Rightarrow \beta\ set$	union over a set
`Inter`	$(\alpha\ set)set \Rightarrow \alpha\ set$	set of sets intersection
`Union`	$(\alpha\ set)set \Rightarrow \alpha\ set$	set of sets union
`Pow`	$\alpha\ set \Rightarrow (\alpha\ set)set$	powerset
`range`	$(\alpha \Rightarrow \beta) \Rightarrow \beta\ set$	range of a function
`Ball Bex`	$[\alpha\ set, \alpha \Rightarrow bool] \Rightarrow bool$	bounded quantifiers

CONSTANTS

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

BINDERS

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

INFIXES

Figure 2.7: Syntax of the theory `Set`

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

TRANSLATIONS

---

term = other terms...
| `{}`
| `{` *term* (`,`*term*)* `}`
| `{` *id* `.` *formula* `}`
| *term* `‘‘` *term*
| *term* `Int` *term*
| *term* `Un` *term*
| `INT` *id*`:`*term* `.` *term*
| `UN` *id*`:`*term* `.` *term*
| `INT` *id* *id** `.` *term*
| `UN` *id* *id** `.` *term*

formula = other formulae...
| *term* `:` *term*
| *term* `~:` *term*
| *term* `<=` *term*
| `ALL` *id*`:`*term* `.` *formula* | `!` *id*`:`*term* `.` *formula*
| `EX` *id*`:`*term* `.` *formula* | `?` *id*`:`*term* `.` *formula*

FULL GRAMMAR

Figure 2.8: Syntax of the theory `Set` (continued)

note that these sets are distinct from those of ZF set theory, and behave more like ZF classes.

- Sets are given by predicates over some type σ. Types serve to define universes for sets, but type-checking is still significant.
- There is a universal set (for each type). Thus, sets have complements, and may be defined by absolute comprehension.
- Although sets may contain other sets as elements, the containing set must have a more complex type.

Finite unions and intersections have the same behaviour in HOL as they do in ZF. In HOL the intersection of the empty set is well-defined, denoting the universal set for the given type.

2.3.1 Syntax of set theory

HOL's set theory is called `Set`. The type $\alpha\,set$ is essentially the same as $\alpha \Rightarrow bool$. The new type is defined for clarity and to avoid complications involving function types in unification. The isomorphisms between the two types are declared explicitly. They are very natural: `Collect` maps $\alpha \Rightarrow bool$ to $\alpha\,set$, while `op :` maps in the other direction (ignoring argument order).

Figure 2.7 lists the constants, infixes, and syntax translations. Figure 2.8 presents the grammar of the new constructs. Infix operators include union and intersection ($A \cup B$ and $A \cap B$), the subset and membership relations, and the image operator ``‘‘``. Note that a`~:`b is translated to $\neg(a \in b)$.

The $\{a_1, \ldots\}$ notation abbreviates finite sets constructed in the obvious manner using `insert` and $\{\}$:

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

The set `{`x`.` $P[x]$`}` consists of all x (of suitable type) that satisfy $P[x]$, where $P[x]$ is a formula that may contain free occurrences of x. This syntax expands to `Collect`$(\lambda x\,.\,P[x])$. It defines sets by absolute comprehension, which is impossible in ZF; the type of x implicitly restricts the comprehension.

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

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

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

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

Figure 2.9: Rules of the theory `Set`

The constants `Ball` and `Bex` are defined accordingly. Instead of `Ball` A P and `Bex` A P we may write `ALL` x`:`A`.` $P[x]$ and `EX` x`:`A`.` $P[x]$. The original notation of Gordon's HOL system is supported as well: `!` and `?`.

Unions and intersections over sets, namely $\bigcup_{x\in A} B[x]$ and $\bigcap_{x\in A} B[x]$, are written `UN` x`:`A`.` $B[x]$ and `INT` x`:`A`.` $B[x]$.

Unions and intersections over types, namely $\bigcup_x B[x]$ and $\bigcap_x B[x]$, are written `UN` x`.` $B[x]$ and `INT` x`.` $B[x]$. They are equivalent to the previous union and intersection operators when A is the universal set.

The operators $\bigcup A$ and $\bigcap A$ act upon sets of sets. They are not binders, but are equal to $\bigcup_{x\in A} x$ and $\bigcap_{x\in A} x$, respectively.

2.3.2 Axioms and rules of set theory

Figure 2.9 presents the rules of theory `Set`. The axioms `mem_Collect_eq` and `Collect_mem_eq` assert that the functions `Collect` and `op :` are isomorphisms. Of course, `op :` also serves as the membership relation.

All the other axioms are definitions. They include the empty set, bounded quantifiers, unions, intersections, complements and the subset relation. They also include straightforward constructions on functions: image (`` `` '') and

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

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

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

Comprehension and Bounded quantifiers

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

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

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

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

The subset and equality relations

Figure 2.10: Derived rules for set theory

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

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

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

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

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

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

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

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

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

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

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

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

Figure 2.11: Further derived rules for set theory

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

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

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

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

Figure 2.12: Derived rules involving subsets

`range`.

Figures 2.10 and 2.11 present derived rules. Most are obvious and resemble rules of Isabelle's ZF set theory. Certain rules, such as `subsetCE`, `bexCI` and `UnCI`, are designed for classical reasoning; the rules `subsetD`, `bexI`, `Un1` and `Un2` are not strictly necessary but yield more natural proofs. Similarly, `equalityCE` supports classical reasoning about extensionality, after the fashion of `iffCE`. See the file `HOL/Set.ML` for proofs pertaining to set theory.

Figure 2.12 presents lattice properties of the subset relation. Unions form least upper bounds; non-empty intersections form greatest lower bounds. Reasoning directly about subsets often yields clearer proofs than reasoning about the membership relation. See the file `HOL/subset.ML`.

Figure 2.13 presents many common set equalities. They include commutative, associative and distributive laws involving unions, intersections and complements. For a complete listing see the file `HOL/equalities.ML`.

! `Blast_tac` proves many set-theoretic theorems automatically. Hence you seldom need to refer to the theorems above.

2.3.3 Properties of functions

Figure 2.14 presents a theory of simple properties of functions. Note that `inv` f uses Hilbert's ε to yield an inverse of f. See the file `HOL/Fun.ML` for a complete listing of the derived rules. Reasoning about function composition (the operator `o`) and the predicate `surj` is done simply by expanding the definitions.

```
Int_absorb        A Int A = A
Int_commute       A Int B = B Int A
Int_assoc         (A Int B) Int C  =  A Int (B Int C)
Int_Un_distrib    (A Un B)  Int C  =  (A Int C) Un (B Int C)

Un_absorb         A Un A = A
Un_commute        A Un B = B Un A
Un_assoc          (A Un B)  Un C  =  A Un (B Un C)
Un_Int_distrib    (A Int B) Un C  =  (A Un C) Int (B Un C)

Compl_disjoint    A Int (-A) = {x. False}
Compl_partition   A Un  (-A) = {x. True}
double_complement -(-A) = A
Compl_Un          -(A Un B)  = (-A) Int (-B)
Compl_Int         -(A Int B) = (-A) Un (-B)

Union_Un_distrib  Union(A Un B) = (Union A) Un (Union B)
Int_Union         A Int (Union B) = (UN C:B. A Int C)

Inter_Un_distrib  Inter(A Un B) = (Inter A) Int (Inter B)
Un_Inter          A Un (Inter B) = (INT C:B. A Un C)
```

Figure 2.13: Set equalities

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

```
inj_def         inj f      == ! x y. f x=f y --> x=y
surj_def        surj f     == ! y. ? x. y=f x
inj_on_def      inj_on f A == !x:A. !y:A. f x=f y --> x=y
inv_def         inv f      == (%y. @x. f(x)=y)
```

Figure 2.14: Theory `Fun`

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

2.4 Simplification and substitution

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

See the *Reference Manual* for details of substitution and simplification.

! Reducing $a = b \wedge P(a)$ to $a = b \wedge P(b)$ is sometimes advantageous. The left part of a conjunction helps in simplifying the right part. This effect is not available by default: it can be slow. It can be obtained by including `conj_cong` in a simpset, `addcongs [conj_cong]`.

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

If the simplifier cannot use a certain rewrite rule — either because of nontermination or because its left-hand side is too flexible — then you might try `stac`:

`stac` *thm* *i*, where *thm* is of the form *lhs* = *rhs*, replaces in subgoal *i* instances of *lhs* by corresponding instances of *rhs*. In case of multiple instances of *lhs* in subgoal *i*, backtracking may be necessary to select the desired ones.

If *thm* is a conditional equality, the instantiated condition becomes an additional (first) subgoal.

HOL provides the tactic `hyp_subst_tac`, which substitutes for an equality throughout a subgoal and its hypotheses. This tactic uses HOL's general substitution rule.

2.4.1 Case splitting

HOL also provides convenient means for case splitting during rewriting. Goals containing a subterm of the form `if` *b* `then...else...` often require

a case distinction on b. This is expressed by the theorem `split_if`:

$$?P(\texttt{if}\ ?b\ \texttt{then}\ ?x\ \texttt{else}\ ?y)\ =\ ((?b \to ?P(?x)) \wedge (\neg ?b \to ?P(?y))) \quad (*)$$

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

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

Because $(*)$ is too general as a rewrite rule for the simplifier (the left-hand side is not a higher-order pattern in the sense of the *Reference Manual*), there is a special infix function `addsplits` of type `simpset * thm list -> simpset` (analogous to `addsimps`) that adds rules such as $(*)$ to a simpset, as in

```
by(simp_tac (simpset() addsplits [split_if]) 1);
```

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

It turns out that using `split_if` is almost always the right thing to do. Hence `split_if` is already included in the default simpset. If you want to delete it from a simpset, use `delsplits`, which is the inverse of `addsplits`:

```
by(simp_tac (simpset() delsplits [split_if]) 1);
```

In general, `addsplits` accepts rules of the form

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

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

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

```
Addsplits: thm list -> unit
Delsplits: thm list -> unit
```

for adding splitting rules to, and deleting them from the current simpset.

2.5 Types

This section describes HOL's basic predefined types ($\alpha \times \beta$, $\alpha + \beta$, *nat* and α *list*) and ways for introducing new types in general. The most important type construction, the `datatype`, is treated separately in §2.6.

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

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

Pair_eq      ((a,b) = (a',b')) = (a=a' & b=b')
Pair_inject  [| (a, b) = (a',b');  [| a=a';  b=b' |] ==> R |] ==> R
PairE        [| !!x y. p = (x,y) ==> Q |] ==> Q

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

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

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

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

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

2.5.1 Product and sum types

Theory `Prod` (Fig. 2.15) defines the product type $\alpha \times \beta$, with the ordered pair syntax (a, b). General tuples are simulated by pairs nested to the right:

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

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

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

Nested patterns are also supported. They are translated stepwise:

$$\begin{aligned} \texttt{\%(}x\texttt{,}y\texttt{,}z\texttt{).}\ t &\leadsto \texttt{\%(}x\texttt{,(}y\texttt{,}z\texttt{)).}\ t \\ &\leadsto \texttt{split(\%}x\texttt{.\%(}y\texttt{,}z\texttt{).}\ t\texttt{)} \\ &\leadsto \texttt{split(\%}x\texttt{.}\ \texttt{split(\%}y\ z\texttt{.}\ t\texttt{))} \end{aligned}$$

The reverse translation is performed upon printing.

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

In addition to explicit λ-abstractions, patterns can be used in any variable binding construct which is internally described by a λ-abstraction. Some important examples are

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

Quantifiers: `ALL` *pattern*$: A$`.` P

Choice: `SOME` *pattern*`.` P

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

Sets: `{`*pattern*`.` P`}`

There is a simple tactic which supports reasoning about patterns:

`split_all_tac` i replaces in subgoal i all `!!`-quantified variables of product type by individual variables for each component. A simple example:

```
  1. !!p. (%(x,y,z). (x, y, z)) p = p
by(split_all_tac 1);
  1. !!x xa ya. (%(x,y,z). (x, y, z)) (x, xa, ya) = (x, xa, ya)
```

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

```
Inl_not_Inr    Inl a ~= Inr b

inj_Inl        inj Inl
inj_Inr        inj Inr

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

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

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

Figure 2.16: Type $\alpha + \beta$

Theory `Prod` also introduces the degenerate product type `unit` which contains only a single element named `()` with the property

```
unit_eq        u = ()
```

Theory `Sum` (Fig. 2.16) defines the sum type $\alpha+\beta$ which associates to the right and has a lower priority than $*$: $\tau_1+\tau_2+\tau_3*\tau_4$ means $\tau_1+(\tau_2+(\tau_3*\tau_4))$.

The definition of products and sums in terms of existing types is not shown. The constructions are fairly standard and can be found in the respective theory files. Although the sum and product types are constructed manually for foundational reasons, they are represented as actual datatypes later.

2.5.2 The type of natural numbers, *nat*

The theory `Nat` defines the natural numbers in a roundabout but traditional way. The axiom of infinity postulates a type *ind* of individuals, which is non-empty and closed under an injective operation. The natural numbers are inductively generated by choosing an arbitrary individual for 0 and using the injective operation to take successors. This is a least fixedpoint construction.

Type *nat* is an instance of class `ord`, which makes the overloaded functions of this class (especially `<` and `<=`, but also `min`, `max` and `LEAST`) available on

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

CONSTANTS AND INFIXES

```
nat_induct     [| P 0; !!n. P n ==> P(Suc n) |]  ==> P n

Suc_not_Zero   Suc m ~= 0
inj_Suc        inj Suc
n_not_Suc_n    n~=Suc n
```

BASIC PROPERTIES

Figure 2.17: The type of natural numbers, *nat*

```
                0+n           = n
                (Suc m)+n     = Suc(m+n)

                m-0           = m
                0-n           = n
                Suc(m)-Suc(n) = m-n

                0*n           = 0
                Suc(m)*n      = n + m*n

mod_less        m<n ==> m mod n = m
mod_geq         [| 0<n;  ~m<n |] ==> m mod n = (m-n) mod n

div_less        m<n ==> m div n = 0
div_geq         [| 0<n;  ~m<n |] ==> m div n = Suc((m-n) div n)
```

Figure 2.18: Recursion equations for the arithmetic operators

nat. Theory `Nat` also shows that `<=` is a linear order, so *nat* is also an instance of class `linorder`.

Theory `NatArith` develops arithmetic on the natural numbers. It defines addition, multiplication and subtraction. Theory `Divides` defines division, remainder and the "divides" relation. The numerous theorems proved include commutative, associative, distributive, identity and cancellation laws. See Figs. 2.17 and 2.18. The recursion equations for the operators `+`, `-` and `*` on `nat` are part of the default simpset.

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

```
primrec
      "0 + n = n"
  "Suc m + n = Suc (m + n)"
```

There is also a `case`-construct of the form

```
case e of 0 => a | Suc m => b
```

Note that Isabelle insists on precisely this format; you may not even change the order of the two cases. Both `primrec` and `case` are realized by a recursion operator `nat_rec`, which is available because *nat* is represented as a datatype.

Tactic `induct_tac "`n`"` i performs induction on variable n in subgoal i using theorem `nat_induct`. There is also the derived theorem `less_induct`:

```
[| !!n. [| ! m. m<n --> P m |] ==> P n |]  ==>  P n
```

2.5.3 Numerical types and numerical reasoning

The integers (type `int`) are also available in HOL, and the reals (type `real`) are available in the logic image `HOL-Complex`. They support the expected operations of addition (`+`), subtraction (`-`) and multiplication (`*`), and much else. Type `int` provides the `div` and `mod` operators, while type `real` provides real division and other operations. Both types belong to class `linorder`, so they inherit the relational operators and all the usual properties of linear orderings. For full details, please survey the theories in subdirectories `Integ`, `Real`, and `Complex`.

All three numeric types admit numerals of the form $sd \ldots d$, where s is an optional minus sign and $d \ldots d$ is a string of digits. Numerals are represented internally by a datatype for binary notation, which allows numerical calculations to be performed by rewriting. For example, the integer division of `54342339` by `3452` takes about five seconds. By default, the simplifier cancels like terms on the opposite sites of relational operators (reducing `z+x<x+y` to

`z<y`, for instance. The simplifier also collects like terms, replacing `x+y+x*3` by `4*x+y`.

Sometimes numerals are not wanted, because for example `n+3` does not match a pattern of the form `Suc` k. You can re-arrange the form of an arithmetic expression by proving (via `subgoal_tac`) a lemma such as `n+3 = Suc (Suc (Suc n))`. As an alternative, you can disable the fancier simplifications by using a basic simpset such as `HOL_ss` rather than the default one, `simpset()`.

Reasoning about arithmetic inequalities can be tedious. Fortunately, HOL provides a decision procedure for *linear arithmetic*: formulae involving addition and subtraction. The simplifier invokes a weak version of this decision procedure automatically. If this is not sufficent, you can invoke the full procedure `Lin_Arith.tac` explicitly. It copes with arbitrary formulae involving `=`, `<`, `<=`, `+`, `-`, `Suc`, `min`, `max` and numerical constants. Other subterms are treated as atomic, while subformulae not involving numerical types are ignored. Quantified subformulae are ignored unless they are positive universal or negative existential. The running time is exponential in the number of occurrences of `min`, `max`, and `-` because they require case distinctions. If `k` is a numeral, then `div k`, `mod k` and `k dvd` are also supported. The former two are eliminated by case distinctions, again blowing up the running time. If the formula involves explicit quantifiers, `Lin_Arith.tac` may take super-exponential time and space.

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

2.5.4 The type constructor for lists, *list*

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

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

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

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

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

Constants and infixes

---

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

Translations

Figure 2.19: The theory `List`

```
null [] = True
null (x#xs) = False

hd (x#xs) = x

tl (x#xs) = xs
tl [] = []

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

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

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

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

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

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

xs!0 = hd xs
xs!(Suc n) = (tl xs)!n
```

Figure 2.20: Simple list processing functions

```
map f [] = []
map f (x#xs) = f x # map f xs

filter P [] = []
filter P (x#xs) = (if P x then x#filter P xs else filter P xs)

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

take n [] = []
take n (x#xs) = (case n of 0 => [] | Suc(m) => x # take m xs)

drop n [] = []
drop n (x#xs) = (case n of 0 => x#xs | Suc(m) => drop m xs)

takeWhile P [] = []
takeWhile P (x#xs) = (if P x then x#takeWhile P xs else [])

dropWhile P [] = []
dropWhile P (x#xs) = (if P x then dropWhile P xs else xs)
```

Figure 2.21: Further list processing functions

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

`List` provides a basic library of list processing functions defined by primitive recursion. The recursion equations are shown in Figs. 2.20 and 2.21.

2.6 Datatype definitions

Inductive datatypes, similar to those of ML, frequently appear in applications of Isabelle/HOL. In principle, such types could be defined by hand via `typedef`, but this would be far too tedious. The `datatype` definition package of Isabelle/HOL (cf. [1]) automates such chores. It generates an appropriate `typedef` based on a least fixed-point construction, and proves freeness theorems and induction rules, as well as theorems for recursion and case combinators. The user just has to give a simple specification of new inductive types using a notation similar to ML or Haskell.

The current datatype package can handle both mutual and indirect recursion. It also offers to represent existing types as datatypes giving the advantage of a more uniform view on standard theories.

2.6.1 Basics

A general `datatype` definition is of the following form:

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

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

- τ is non-recursive, i.e. τ does not contain any of the newly defined type constructors $t_1, \ldots, t_n$, or
- $\tau = (\vec{\alpha})t_{j'}$ where $1 \leq j' \leq n$, or
- $\tau = (\tau_1', \ldots, \tau_{h'}')t'$, where t' is the type constructor of an already existing datatype and $\tau_1', \ldots, \tau_{h'}'$ are admissible types.
- $\tau = \sigma \to \tau'$, where τ' is an admissible type and σ is non-recursive (i.e. the occurrences of the newly defined types are *strictly positive*)

If some $(\vec{\alpha})t_{j'}$ occurs in a type $\tau_{i,i'}^j$ of the form

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

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

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

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

```
datatype 'a aexp = If_then_else ('a bexp) ('a aexp) ('a aexp)
                 | Sum ('a aexp) ('a aexp)
                 | Diff ('a aexp) ('a aexp)
                 | Var 'a
                 | Num nat
and      'a bexp = Less ('a aexp) ('a aexp)
                 | And ('a bexp) ('a bexp)
                 | Or ('a bexp) ('a bexp)
```

The datatype `term`, which is defined by

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

is an example for a datatype with nested recursion. Using nested recursion involving function spaces, we may also define infinitely branching datatypes, e.g.

```
datatype 'a tree = Atom 'a | Branch "nat => 'a tree"
```

Types in HOL must be non-empty. Each of the new datatypes $(\vec{\alpha})t_j$ with $1 \leq j \leq n$ is non-empty if and only if it has a constructor C_i^j with the following property: for all argument types $\tau_{i,i'}^j$ of the form $(\vec{\alpha})t_{j'}$ the datatype $(\vec{\alpha})t_{j'}$ is non-empty.

If there are no nested occurrences of the newly defined datatypes, obviously at least one of the newly defined datatypes $(\vec{\alpha})t_j$ must have a constructor C_i^j without recursive arguments, a *base case*, to ensure that the new types are non-empty. If there are nested occurrences, a datatype can even be non-empty without having a base case itself. Since `list` is a non-empty datatype, `datatype t = C (t list)` is non-empty as well.

Freeness of the constructors

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

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

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

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

The constructor functions are injective:

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

Since the number of distinctness inequalities is quadratic in the number of constructors, the datatype package avoids proving them separately if there are too many constructors. Instead, specific inequalities are proved by a suitable simplification procedure on demand.[3]

[3]This procedure, which is already part of the default simpset, may be referred to by the ML identifier `DatatypePackage.distinct_simproc`.

Structural induction

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

$$\frac{\begin{array}{lcl}
\bigwedge x_1 \ldots x_{m_1^1} . [\![P_{s_{1,1}^1}\ x_{r_{1,1}^1}; \ldots; P_{s_{1,l_1^1}^1}\ x_{r_{1,l_1^1}^1}]\!] & \Longrightarrow & P_1\ \left(C_1^1\ x_1\ \ldots\ x_{m_1^1}\right) \\
\vdots & & \\
\bigwedge x_1 \ldots x_{m_{k_1}^1} . [\![P_{s_{k_1,1}^1}\ x_{r_{k_1,1}^1}; \ldots; P_{s_{k_1,l_{k_1}^1}^1}\ x_{r_{k_1,l_{k_1}^1}^1}]\!] & \Longrightarrow & P_1\ \left(C_{k_1}^1\ x_1\ \ldots\ x_{m_{k_1}^1}\right) \\
\vdots & & \\
\bigwedge x_1 \ldots x_{m_1^n} . [\![P_{s_{1,1}^n}\ x_{r_{1,1}^n}; \ldots; P_{s_{1,l_1^n}^n}\ x_{r_{1,l_1^n}^n}]\!] & \Longrightarrow & P_n\ \left(C_1^n\ x_1\ \ldots\ x_{m_1^n}\right) \\
\vdots & & \\
\bigwedge x_1 \ldots x_{m_{k_n}^n} . [\![P_{s_{k_n,1}^n}\ x_{r_{k_n,1}^n}; \ldots P_{s_{k_n,l_{k_n}^n}^n}\ x_{r_{k_n,l_{k_n}^n}^n}]\!] & \Longrightarrow & P_n\ \left(C_{k_n}^n\ x_1\ \ldots\ x_{m_{k_n}^n}\right)
\end{array}}{P_1\ x_1 \wedge \ldots \wedge P_n\ x_n}$$

where

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

i.e. the properties P_j can be assumed for all recursive arguments.

For datatypes with nested recursion, such as the `term` example from above, things are a bit more complicated. Conceptually, Isabelle/HOL unfolds a definition like

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

to an equivalent definition without nesting:

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

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

$$\frac{\begin{array}{l}
\bigwedge x\ .\ P_1\ (\texttt{Var}\ x) \\
\bigwedge x_1\ x_2\ .\ P_2\ x_2 \Longrightarrow P_1\ (\texttt{App}\ x_1\ x_2) \\
P_2\ \texttt{Nil} \\
\bigwedge x_1\ x_2\ .\ [\![P_1\ x_1; P_2\ x_2]\!] \Longrightarrow P_2\ (\texttt{Cons}\ x_1\ x_2)
\end{array}}{P_1\ x_1 \wedge P_2\ x_2}$$

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

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

$$\frac{\bigwedge a\,.\;\; P\;(\texttt{Atom}\;a) \quad \bigwedge ts\,.\;\; (\forall x\,.\;\; P\;(ts\;x)) \Longrightarrow P\;(\texttt{Branch}\;ts)}{P\;t}$$

In principle, inductive types are already fully determined by freeness and structural induction. For convenience in applications, the following derived constructions are automatically provided for any datatype.

The `case` construct

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

$$\begin{array}{lll} \texttt{case}\;e\;\texttt{of} & C_1^j\;x_{1,1}\;\ldots\;x_{1,m_1^j} & \Rightarrow\;\; e_1 \\ \vdots & & \\ \mid & C_{k_j}^j\;x_{k_j,1}\;\ldots\;x_{k_j,m_{k_j}^j} & \Rightarrow\;\; e_{k_j} \end{array}$$

where the $x_{i,j}$ are either identifiers or nested tuple patterns as in §2.5.1.

! All constructors must be present, their order is fixed, and nested patterns are not supported (with the exception of tuples). Violating this restriction results in strange error messages.

To perform case distinction on a goal containing a `case`-construct, the theorem t_j`.split` is provided:

$$\begin{array}{l} P(t_j\texttt{\_case}\;f_1\;\ldots\;f_{k_j}\;e) = ((\forall x_1 \ldots x_{m_1^j}\,.\, e = C_1^j\;x_1 \ldots x_{m_1^j} \rightarrow P(f_1\;x_1 \ldots x_{m_1^j})) \\ \qquad\qquad \wedge\;\; \ldots\;\; \wedge \\ \qquad\qquad (\forall x_1 \ldots x_{m_{k_j}^j}\,.\, e = C_{k_j}^j\;x_1 \ldots x_{m_{k_j}^j} \rightarrow P(f_{k_j}\;x_1 \ldots x_{m_{k_j}^j}))) \end{array}$$

where t_j`_case` is the internal name of the `case`-construct. This theorem can be added to a simpset via `addsplits` (see §2.4.1).

Case splitting on assumption works as well, by using the rule t_j`.split_asm` in the same manner. Both rules are available under t_j`.splits` (this name is *not* bound in ML, though).

! By default only the selector expression (e above) in a `case`-construct is simplified, in analogy with `if` (see page 20). Only if that reduces to a constructor is one of the arms of the `case`-construct exposed and simplified. To ensure full simplification of all parts of a `case`-construct for datatype t, remove t`.case_weak_cong` from the simpset, for example by `delcongs [thm "`t`.weak_case_cong"]`.

The function `size`

Theory `NatArith` declares a generic function `size` of type $\alpha \Rightarrow nat$. Each datatype defines a particular instance of `size` by overloading according to the following scheme:

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

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

2.6.2 Defining datatypes

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

Most of the theorems about datatypes become part of the default simpset and you never need to see them again because the simplifier applies them automatically. Only induction or case distinction are usually invoked by hand.

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

`induct_tac "`$x_1\ \ldots\ x_n$`"` i applies simultaneous structural induction on the variables $x_1, \ldots, x_n$ to subgoal i. This is the canonical way to prove properties of mutually recursive datatypes such as `aexp` and `bexp`, or datatypes with nested recursion such as `term`.

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

`case_tac "`u`"` i performs a case analysis for the term u whose type must be a datatype. If the datatype has k_j constructors $C_1^j, \ldots C_{k_j}^j$, subgoal i is replaced by k_j new subgoals which contain the additional assumption $u = C_{i'}^j\ x_1\ \ldots\ x_{m_{i'}^j}$ for $i' = 1, \ldots, k_j$.

Note that induction is only allowed on free variables that should not occur among the premises of the subgoal. Case distinction applies to arbitrary terms.

For the technically minded, we exhibit some more details. Processing the theory file produces an ML structure which, in addition to the usual components, contains a structure named t for each datatype t defined in the file. Each structure t contains the following elements:

```
val distinct : thm list
val inject : thm list
val induct : thm
val exhaust : thm
val cases : thm list
val split : thm
val split_asm : thm
val recs : thm list
val size : thm list
val simps : thm list
```

`distinct`, `inject`, `induct`, `size` and `split` contain the theorems described above. For user convenience, `distinct` contains inequalities in both directions. The reduction rules of the `case`-construct are in `cases`. All theorems from `distinct`, `inject` and `cases` are combined in `simps`. In case of mutually recursive datatypes, `recs`, `size`, `induct` and `simps` are contained in a separate structure named $t_1\_\ldots\_t_n$.

2.7 Old-style recursive function definitions

Old-style recursive definitions via `recdef` requires that you supply a well-founded relation that governs the recursion. Recursive calls are only allowed if they make the argument decrease under the relation. Complicated recursion forms, such as nested recursion, can be dealt with. Termination can even be proved at a later time, though having unsolved termination conditions around can make work difficult.[4]

Using `recdef`, you can declare functions involving nested recursion and pattern-matching. Recursion need not involve datatypes and there are few syntactic restrictions. Termination is proved by showing that each recursive call makes the argument smaller in a suitable sense, which you specify by supplying a well-founded relation.

Here is a simple example, the Fibonacci function. The first line declares `fib` to be a constant. The well-founded relation is simply $<$ (on the natural numbers). Pattern-matching is used here: `1` is a macro for `Suc 0`.

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

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

With `recdef`, function definitions may be incomplete, and patterns may overlap, as in functional programming. The `recdef` package disambiguates overlapping patterns by taking the order of rules into account. For missing patterns, the function is defined to return a default value.

The well-founded relation defines a notion of "smaller" for the function's argument type. The relation $\prec$ is **well-founded** provided it admits no infinitely decreasing chains

$$\cdots \prec x_n \prec \cdots \prec x_1.$$

If the function's argument has type τ, then $\prec$ has to be a relation over τ: it must have type $(\tau \times \tau)set$.

Proving well-foundedness can be tricky, so Isabelle/HOL provides a collection of operators for building well-founded relations. The package recognises these operators and automatically proves that the constructed relation is well-founded. Here are those operators, in order of importance:

- `less_than` is "less than" on the natural numbers. (It has type $(nat \times nat)set$, while $<$ has type $[nat, nat] \Rightarrow bool$.
- `measure`f, where f has type $\tau \Rightarrow nat$, is the relation $\prec$ on type τ such that $x \prec y$ if and only if $f(x) < f(y)$. Typically, f takes the recursive function's arguments (as a tuple) and returns a result expressed in terms of the function `size`. It is called a **measure function**. Recall that `size` is overloaded and is defined on all datatypes (see §2.6.1).
- `inv_image`$R\ f$ is a generalisation of `measure`. It specifies a relation such that $x \prec y$ if and only if $f(x)$ is less than $f(y)$ according to R, which must itself be a well-founded relation.
- R_1`<*lex*>`R_2 is the lexicographic product of two relations. It is a relation on pairs and satisfies $(x_1, x_2) \prec (y_1, y_2)$ if and only if x_1 is less than y_1 according to R_1 or $x_1 = y_1$ and x_2 is less than y_2 according to R_2.
- `finite_psubset` is the proper subset relation on finite sets.

We can use `measure` to declare Euclid's algorithm for the greatest common divisor. The measure function, $\lambda(m, n)\ .\ n$, specifies that the recursion terminates because argument n decreases.

```
recdef gcd "measure ((%(m,n). n) ::nat*nat=>nat)"
    "gcd (m, n) = (if n=0 then m else gcd(n, m mod n))"
```

The general form of a well-founded recursive definition is

```
recdef function rel
    congs     congruence rules      (optional)
    simpset   simplification set    (optional)
    reduction rules
```

where

- *function* is the name of the function, either as an *id* or a *string*.
- *rel* is a HOL expression for the well-founded termination relation.
- *congruence rules* are required only in highly exceptional circumstances.
- The *simplification set* is used to prove that the supplied relation is well-founded. It is also used to prove the **termination conditions**: assertions that arguments of recursive calls decrease under *rel*. By default, simplification uses `simpset()`, which is sufficient to prove well-foundedness for the built-in relations listed above.
- *reduction rules* specify one or more recursion equations. Each left-hand side must have the form $f\ t$, where f is the function and t is a tuple of distinct variables. If more than one equation is present then f is defined by pattern-matching on components of its argument whose type is a `datatype`.

  The ML identifier *f*`.simps` contains the reduction rules as a list of theorems.

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

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

The theory `HOL/ex/Primes` illustrates how to prove termination conditions afterwards. The function `Tfl.tgoalw` is like the standard function `goalw`, which sets up a goal to prove, but its argument should be the identifier *f*`.simps` and its effect is to set up a proof of the termination conditions:

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

This subgoal has a one-step proof using `simp_tac`. Once the theorem is proved, it can be used to eliminate the termination conditions from elements of `gcd.simps`. Theory `HOL/Subst/Unify` is a much more complicated example of this process, where the termination conditions can only be proved by complicated reasoning involving the recursive function itself.

Isabelle/HOL can prove the `gcd` function's termination condition automatically if supplied with the right simpset.

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

If all termination conditions were proved automatically, f`.simps` is added to the simpset automatically, just as in `primrec`. The simplification rules corresponding to clause i (where counting starts at 0) are called $f.i$ and can be accessed as `thms "`$f.i$`"`, which returns a list of theorems. Thus you can, for example, remove specific clauses from the simpset. Note that a single clause may give rise to a set of simplification rules in order to capture the fact that if clauses overlap, their order disambiguates them.

A `recdef` definition also returns an induction rule specialised for the recursive function. For the `gcd` function above, the induction rule is

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

This rule should be used to reason inductively about the `gcd` function. It usually makes the induction hypothesis available at all recursive calls, leading to very direct proofs. If any termination conditions remain unproved, they will become additional premises of this rule.

2.8 Example: Cantor's Theorem

Cantor's Theorem states that every set has more subsets than it has elements. It has become a favourite example in higher-order logic since it is so easily expressed:

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

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

The Isabelle proof uses HOL's set theory, with the type $\alpha\,set$ and the operator `range`.

```
context Set.thy;
```

The set S is given as an unknown instead of a quantified variable so that we may inspect the subset found by the proof.

```
Goal "?S ~: range(f :: 'a=>'a set)";
  Level 0
  ?S ~: range f
   1. ?S ~: range f
```

The first two steps are routine. The rule `rangeE` replaces $?S \in \texttt{range}\,f$ by $?S = f\ x$ for some x.

```
by (resolve_tac [notI] 1);
  Level 1
  ?S ~: range f
   1. ?S : range f ==> False
by (eresolve_tac [rangeE] 1);
  Level 2
  ?S ~: range f
   1. !!x. ?S = f x ==> False
```

Next, we apply `equalityCE`, reasoning that since $?S = f\ x$, we have $?c \in ?S$ if and only if $?c \in f\ x$ for any $?c$.

```
by (eresolve_tac [equalityCE] 1);
  Level 3
  ?S ~: range f
   1. !!x. [| ?c3 x : ?S; ?c3 x : f x |] ==> False
   2. !!x. [| ?c3 x ~: ?S; ?c3 x ~: f x |] ==> False
```

Now we use a bit of creativity. Suppose that $?S$ has the form of a comprehension. Then $?c \in \{x\,.\,?P\ x\}$ implies $?P\ ?c$. Destruct-resolution using `CollectD` instantiates $?S$ and creates the new assumption.

```
by (dresolve_tac [CollectD] 1);
  Level 4
  {x. ?P7 x} ~: range f
   1. !!x. [| ?c3 x : f x; ?P7(?c3 x) |] ==> False
   2. !!x. [| ?c3 x ~: {x. ?P7 x}; ?c3 x ~: f x |] ==> False
```

Forcing a contradiction between the two assumptions of subgoal 1 completes the instantiation of S. It is now the set $\{x\,.\,x \notin f\ x\}$, which is the standard diagonal construction.

```
by (contr_tac 1);
  Level 5
  {x. x ~: f x} ~: range f
   1. !!x. [| x ~: {x. x ~: f x}; x ~: f x |] ==> False
```

The rest should be easy. To apply `CollectI` to the negated assumption, we employ `swap_res_tac`:

```
by (swap_res_tac [CollectI] 1);
  Level 6
  {x. x ~: f x} ~: range f
   1. !!x. [| x ~: f x; ~ False |] ==> x ~: f x
by (assume_tac 1);
  Level 7
  {x. x ~: f x} ~: range f
  No subgoals!
```

How much creativity is required? As it happens, Isabelle can prove this theorem automatically. The default classical set `claset()` contains rules for most of the constructs of HOL's set theory. We must augment it with `equalityCE` to break up set equalities, and then apply best-first search. Depth-first search would diverge, but best-first search successfully navigates through the large search space.

```
choplev 0;
  Level 0
  ?S ~: range f
   1. ?S ~: range f
by (best_tac (claset() addSEs [equalityCE]) 1);
  Level 1
  {x. x ~: f x} ~: range f
  No subgoals!
```

If you run this example interactively, make sure your current theory contains theory `Set`, for example by executing `context Set.thy`. Otherwise the default claset may not contain the rules for set theory.

Bibliography

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

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

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

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

Index