# Isabelle's Logics: HOL[1]

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

19 April 2009

[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 $\lor$, which has lower priority than $\land$. There, $P \land Q \lor R$ abbreviates $(P \land Q) \lor R$ rather than $P \land (Q \lor R)$. Also, $\exists x\,.\,P \lor Q$ abbreviates $\exists x\,.\,(P \lor Q)$ rather than $(\exists x\,.\,P) \lor Q$. Note especially that $P \lor (\exists x\,.\,Q)$ becomes syntactically invalid if the brackets are removed.

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

The theory `HOL` implements higher-order logic. It is based on Gordon's HOL system [6], which itself is based on Church's original paper [4]. Andrews's book [1] is a full description of the original Church-style higher-order logic. Experience with the HOL system has demonstrated that higher-order logic is widely applicable in many areas of mathematics and computer science, not just hardware verification, HOL's original *raison d'être*. It is weaker than ZF set theory but for most applications this does not matter. If you prefer ML to Lisp, you will probably prefer HOL to ZF.

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

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

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

## 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 $\neg(a = b)$.

[1]Earlier versions of Isabelle's HOL used a different syntax. Ancient releases of Isabelle included still another version of HOL, with explicit type inference rules [18]. This version no longer exists, but `ZF` supports a similar style of reasoning.

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

Constants

| *symbol* | *name* | *meta-type* | *description* |
|---|---|---|---|
| `SOME` or `@` | `Eps` | $(\alpha \Rightarrow bool) \Rightarrow \alpha$ | Hilbert description ($\varepsilon$) |
| `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

! 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 $\sigma$ and $\tau$ do, allowing quantification over functions.

HOL allows new types to be declared as subsets of existing types; see §2.5. ML-like datatypes can also be declared; 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 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 $\alpha :: \{\mathit{ord}, \mathit{plus}\}$, 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 $\tau$ is a type of class `ord`, $P$ a formula and $x$ a variable of type $\tau$, 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 $\varepsilon$ choice operator, so `Least` is always meaningful, but may yield nothing useful in case there is not a unique least element satisfying $P$.[2]

[2]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

All these binders have priority 10.

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

### 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

`case` $e$ `of` $c_1$ `=>` $e_1$ `|` $\ldots$ `|` $c_n$ `=>` $e_n$

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

## 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:

```
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

`ext` expresses extensionality of functions.

`iff` asserts that logically equivalent formulae are equal.

`someI` gives the defining property of the Hilbert $\varepsilon$-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.[3]

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 [6, page 270] using object-equality (`=`), which is possible because equality in higher-order logic may equate formulae and even functions over formulae. But theory HOL, like all other Isabelle theories, uses meta-equality (`==`) for definitions.

! The definitions above should never be expanded and are shown for completeness only. Instead users should reason in terms of the derived rules shown below or, better still, using high-level tactics (see §2.4).

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

[3]In fact, the $\varepsilon$-operator already makes the logic classical, as shown by Diaconescu; see Paulson [18] for details.

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

Equality

```
TrueI        True
FalseE       False ==> P

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

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

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

Propositional logic

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

Logical equivalence

Figure 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

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 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 $\sigma$. 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.

| *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(`$\lambda x . P[x]$`)` | comprehension |
| `INT` $x$:$A$`.` $B[x]$ | `INTER` $A$ $\lambda x . B[x]$ | intersection |
| `UN` $x$:$A$`.` $B[x]$ | `UNION` $A$ $\lambda x . B[x]$ | union |
| `ALL` $x$:$A$`.` $P[x]$ or `!` $x$:$A$`.` $P[x]$ | `Ball` $A$ $\lambda x .$ $P[x]$ | bounded $\forall$ |
| `EX` $x$:$A$`.` $P[x]$ or `?` $x$:$A$`.` $P[x]$ | `Bex` $A$ $\lambda x .$ $P[x]$ | bounded $\exists$ |

Translations

| | | |
|---|---|---|
| *term* | = | other terms... |
| | \| | `{}` |
| | \| | `{` *term* (`,`*term*)$^*$ `}` |
| | \| | `{` *id* `.` *formula* `}` |
| | \| | *term* `` ` ` `` *term* |
| | \| | *term* `Int` *term* |
| | \| | *term* `Un` *term* |
| | \| | `INT` *id*:*term* `.` *term* |
| | \| | `UN` *id*:*term* `.` *term* |
| | \| | `INT` *id* *id*$^*$ `.` *term* |
| | \| | `UN` *id* *id*$^*$ `.` *term* |
| *formula* | = | other formulae... |
| | \| | *term* `:` *term* |
| | \| | *term* `~:` *term* |
| | \| | *term* `<=` *term* |
| | \| | `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)

### 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**:

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

The constants `Ball` and `Bex` are defined accordingly. Instead of `Ball` $A$ $P$ and `Bex` $A$ $P$ we may write `ALL` $x$`:`$A$`.` $P[x]$ and `EX` $x$`:`$A$`.` $P[x]$. 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.

```
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`

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

Figures 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.

```
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

```
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`

### 2.3.3 Properties of functions

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

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

## 2.4 Generic packages

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

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

! 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.

## 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`:

$$?P(\texttt{if } ?b \texttt{ then } ?x \texttt{ else } ?y) \;=\; ((?b \to ?P(?x)) \land (\neg ?b \to ?P(?y))) \qquad (*)$$

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

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

Because $(*)$ is too general as a rewrite rule for the simplifier (the left-hand side is not a higher-order pattern in the sense of the *Reference Manual*), there is a special infix function `addsplits` 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

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

where $c$ is a constant and *rhs* is arbitrary. Note that $(*)$ is of the right form because internally the left-hand side is $?P($`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.

### 2.4.2 Classical reasoning

HOL derives classical introduction rules for $\lor$ and $\exists$, as well as classical elimination rules for $\to$ and $\leftrightarrow$, and the swap rule; recall Fig. 2.6 above.

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

## 2.5 Types

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

### 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 |
|---|---|
| $\tau_1 \times \dots \times \tau_n$ | $\tau_1 \times (\dots(\tau_{n-1} \times \tau_n)\dots)$ |
| $(t_1, \dots, t_n)$ | $(t_1, (\dots, (t_{n-1}, t_n)\dots)$ |

| *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 $\alpha\times\beta$

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

`%(`$x$`,`$y$`).` $t$ stands for `split(%`$x\ y$`.` $t$`)`

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 $\lambda$-abstractions, patterns can be used in any variable binding construct which is internally described by a $\lambda$-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)
```

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

| *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 $\alpha + \beta$

```
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 (see §2.6.3). Therefore, the theory `Datatype` should be used instead of `Sum` or `Prod`.

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

| *symbol* | *meta-type* | *priority* | *description* |
|---|---|---|---|
| `0` | $\alpha$ | | 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

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 (see §2.6.3).

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 `linear_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, `linear_arith_tac` may take super-exponential time and space.

If `linear_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 $\alpha$ *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`

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

| *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 (see §2.7.1). The recursion equations are shown in Figs. 2.20 and 2.21.

## 2.5.5 Introducing new types

The HOL-methodology dictates that all extensions to a theory should be **definitional**. The type definition mechanism that meets this criterion is `typedef`. Note that *type synonyms*, which are inherited from Pure and described elsewhere, are just syntactic abbreviations that have no logical meaning.

! Types in HOL must be non-empty; otherwise the quantifier rules would be unsound, because $\exists x \, . \, x = x$ is a theorem [18, §7].

A **type definition** identifies the new type with a subset of an existing type. More precisely, the new type is defined by exhibiting an existing type $\tau$, a set $A :: \tau\ set$, and a theorem of the form $x : A$. Thus $A$ is a non-empty subset of $\tau$, and the new type denotes this subset. New functions are defined that establish an isomorphism between the new type and the subset. If type $\tau$

involves type variables $\alpha_1$, …, $\alpha_n$, then the type definition creates a type constructor $(\alpha_1, \ldots, \alpha_n)ty$ rather than a particular type.

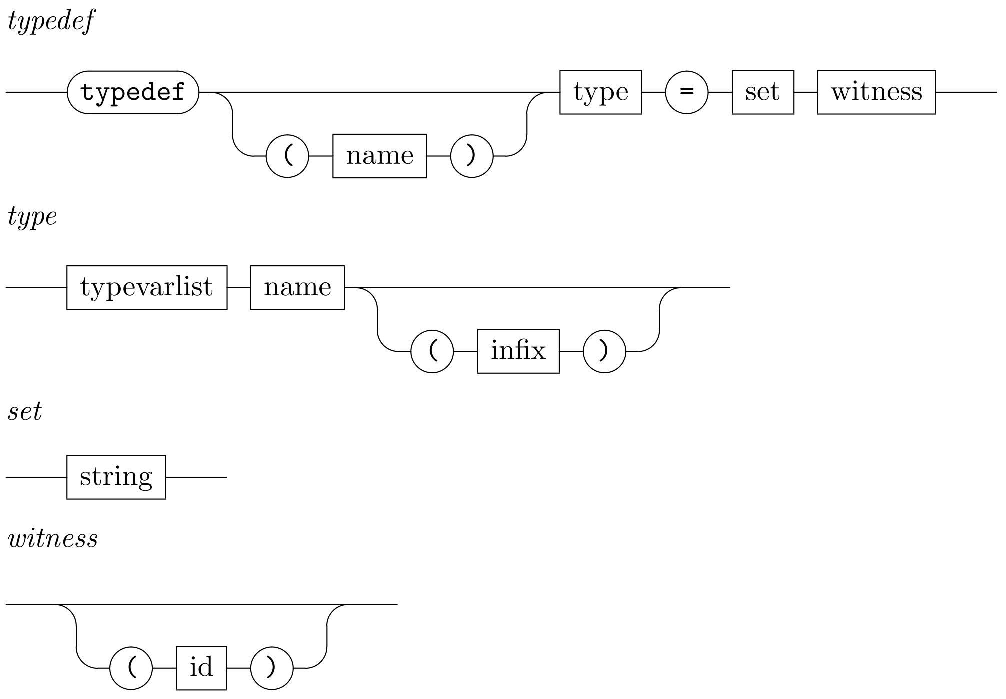

Figure 2.22: Syntax of type definitions

The syntax for type definitions is shown in Fig. 2.22. For the definition of ‘typevarlist’ and ‘infix’ see the appendix of the *Reference Manual*. The remaining nonterminals have the following meaning:

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

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

*set:* the representing subset $A$.

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

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

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

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

- a definition and three axioms

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

  stating that $(\alpha_1, \ldots, \alpha_n)ty$ is isomorphic to $A$ by $Rep\_T$ and its inverse $Abs\_T$.

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

```
typedef unit = "{True}"

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

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

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

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

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

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

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

## 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` (see §2.5.5), but this would be far too tedious. The `datatype` definition package of Isabelle/HOL (cf. [3]) 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:

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

where $\vec{\alpha} = (\alpha_1, \dots, \alpha_h)$ 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 $\alpha_1, \dots, \alpha_h$. A type $\tau$ occurring in a `datatype` definition is *admissible* if and only if

- $\tau$ is non-recursive, i.e. $\tau$ does not contain any of the newly defined type constructors $t_1, \dots, t_n$, or
- $\tau = (\vec{\alpha})t_{j'}$ where $1 \leq j' \leq n$, or
- $\tau = (\tau'_1, \dots, \tau'_{h'})t'$, where $t'$ is the type constructor of an already existing datatype and $\tau'_1, \dots, \tau'_{h'}$ are admissible types.
- $\tau = \sigma \to \tau'$, where $\tau'$ is an admissible type and $\sigma$ 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

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

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.[4]

## Structural induction

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

$$\frac{\begin{array}{lcl}
\bigwedge x_1 \ldots x_{m_1^1} .\ [\![P_{s^1_{1,1}}\ x_{r^1_{1,1}}; \ldots ; P_{s^1_{1,l^1_1}}\ x_{r^1_{1,l^1_1}}]\!] & \Longrightarrow & P_1\ \left(C_1^1\ x_1\ \ldots\ x_{m_1^1}\right) \\
 & \vdots & \\
\bigwedge x_1 \ldots x_{m_{k_1}^1} .\ [\![P_{s^1_{k_1,1}}\ x_{r^1_{k_1,1}}; \ldots ; P_{s^1_{k_1,l^1_{k_1}}}\ x_{r^1_{k_1,l^1_{k_1}}}]\!] & \Longrightarrow & P_1\ \left(C_{k_1}^1\ x_1\ \ldots\ x_{m_{k_1}^1}\right) \\
 & \vdots & \\
\bigwedge x_1 \ldots x_{m_1^n} .\ [\![P_{s^n_{1,1}}\ x_{r^n_{1,1}}; \ldots ; P_{s^n_{1,l^n_1}}\ x_{r^n_{1,l^n_1}}]\!] & \Longrightarrow & P_n\ \left(C_1^n\ x_1\ \ldots\ x_{m_1^n}\right) \\
 & \vdots & \\
\bigwedge x_1 \ldots x_{m_{k_n}^n} .\ [\![P_{s^n_{k_n,1}}\ x_{r^n_{k_n,1}}; \ldots P_{s^n_{k_n,l^n_{k_n}}}\ x_{r^n_{k_n,l^n_{k_n}}}]\!] & \Longrightarrow & P_n\ \left(C_{k_n}^n\ x_1\ \ldots\ x_{m_{k_n}^n}\right)
\end{array}}{P_1\ x_1 \wedge \ldots \wedge P_n\ x_n}$$

where

$$\begin{array}{rcl}
Rec_i^j & := & \left\{\left(r^j_{i,1}, s^j_{i,1}\right), \ldots, \left(r^j_{i,l_i^j}, s^j_{i,l_i^j}\right)\right\} = \\
 & & \left\{(i', i'')\ \middle|\ 1 \le i' \le m_i^j \wedge 1 \le i'' \le n \wedge \tau^j_{i,i'} = (\alpha_1, \ldots, \alpha_h) t_{i''}\right\}
\end{array}$$

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:

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

```
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

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

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

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

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:

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

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:

$$\begin{aligned} P(t_j\texttt{\_case}\ f_1\ \dots\ f_{k_j}\ e) = ((\forall x_1 \dots x_{m_1^j}\,.\ e = C_1^j\ x_1 \dots x_{m_1^j} \to P(f_1\ x_1 \dots x_{m_1^j})) \\ \land\ \dots\ \land \\ (\forall x_1 \dots x_{m_{k_j}^j}\,.\ e = C_{k_j}^j\ x_1 \dots x_{m_{k_j}^j} \to P(f_{k_j}\ x_1 \dots x_{m_{k_j}^j}))) \end{aligned}$$

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 21). 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:

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

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 shown in Fig. 2.23. 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.

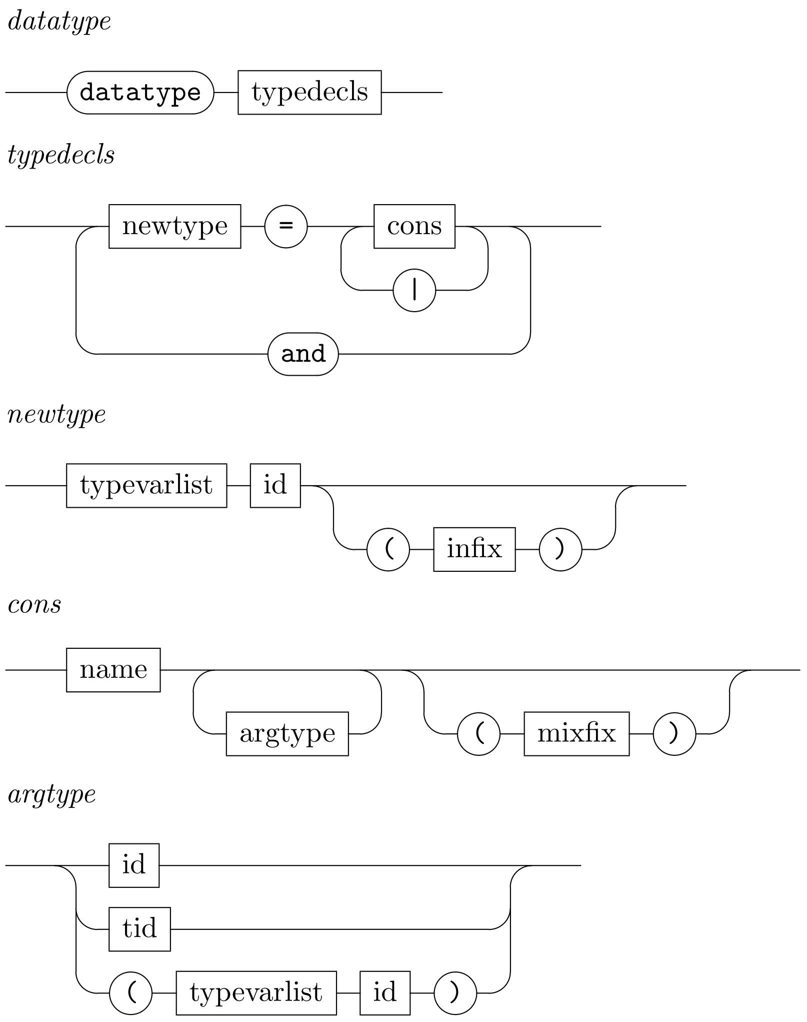

Figure 2.23: Syntax of datatype declarations

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

`induct_tac` "$x_1$ ... $x_n$" $i$ 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 $C_1^j$, ... $C_{k_j}^j$, 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 $t_1$_..._$t_n$.

### 2.6.3 Representing existing types as datatypes

For foundational reasons, some basic types such as `nat`, `*`, `+`, `bool` and `unit` are not defined in a `datatype` section, but by more primitive means using `typedef`. To be able to use the tactics `induct_tac` and `case_tac` and to define functions by primitive recursion on these types, such types may be represented as actual datatypes. This is done by specifying the constructors of the desired type, plus a proof of the induction rule, as well as theorems stating the distinctness and injectivity of constructors in a `rep_datatype` section. For the sum type this works as follows:

```
rep_datatype (sum) Inl Inr
proof -
  fix P
  fix s :: "'a + 'b"
  assume x: "!!x::'a. P (Inl x)" and y: "!!y::'b. P (Inr y)"
  then show "P s" by (auto intro: sumE [of s])
qed simp_all
```

The datatype package automatically derives additional theorems for recursion and case combinators from these rules. Any of the basic HOL types mentioned above are represented as datatypes. Try an induction on `bool` today.

### 2.6.4 Examples

#### The datatype $\alpha$ *mylist*

We want to define a type $\alpha$ *mylist*. To do this we have to build a new theory that contains the type definition. We start from the theory `Datatype` instead of `Main` in order to avoid clashes with the `List` theory of Isabelle/HOL.

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

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

```
Goal "!x. Cons x xs ~= xs";
  Level 0
  ! x. Cons x xs ~= xs
   1. ! x. Cons x xs ~= xs
```

This can be proved by the structural induction tactic:

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

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

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

Similarly, we prove the remaining goal.

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

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

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

### The datatype $\alpha$ *mylist* with mixfix syntax

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

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

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

### A datatype for weekdays

This example shows a datatype that consists of 7 constructors:

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

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

```
Goal "Mon ~= Tue";
by (Simp_tac 1);
```

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

## 2.7 Recursive function definitions

Isabelle/HOL provides two main mechanisms of defining recursive functions.

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

2. **Well-founded recursion** requires that you supply a well-founded relation that governs the recursion. Recursive calls are only allowed if they make the argument decrease under the relation. Complicated recursion forms, such as nested recursion, can be dealt with. Termination can even be proved at a later time, though having unsolved termination conditions around can make work difficult.[5]

Following good HOL tradition, these declarations do not assert arbitrary axioms. Instead, they define the function using a recursion operator. Both HOL and ZF derive the theory of well-founded recursion from first principles [15]. Primitive recursion over some datatype relies on the recursion operator provided by the datatype package. With either form of function definition, Isabelle proves the desired recursion equations as theorems.

[5]This facility is based on Konrad Slind's TFL package [21]. Thanks are due to Konrad for implementing TFL and assisting with its installation.

### 2.7.1 Primitive recursive functions

Datatypes come with a uniform way of defining functions, **primitive recursion**. In principle, one could introduce primitive recursive functions by asserting their reduction rules as new axioms, but this is not recommended:

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

Asserting axioms brings the danger of accidentally asserting nonsense, as in `app [] ys = us`.

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

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

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

```
primrec
    "app [] ys = us"
```

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

The general form of a primitive recursive definition is

```
primrec
    reduction rules
```

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

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

such that $C$ is a constructor of the datatype, $r$ contains only the free variables on the left-hand side, and all recursive calls in $r$ are of the form $f\, \dots\, y_i\, \dots$ for some $i$. There must be at most one reduction rule for each constructor. The order is immaterial. For missing constructors, the function is defined to return a default value.

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

The primitive recursive function can have infix or mixfix syntax:

```
consts "@"  :: ['a list, 'a list] => 'a list  (infixr 60)
primrec
   "[] @ ys = ys"
   "(x#xs) @ ys = x#(xs @ ys)"
```

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

```
Goal "(xs @ ys) @ zs = xs @ (ys @ zs)";
by (induct_tac "xs" 1);
by (ALLGOALS Asm_simp_tac);
```

**Example: Evaluation of expressions**

Using mutual primitive recursion, we can define evaluation functions `evala` and `eval_bexp` for the datatypes of arithmetic and boolean expressions mentioned in §2.6.1:

```
consts
  evala :: "['a => nat, 'a aexp] => nat"
  evalb :: "['a => nat, 'a bexp] => bool"

primrec
  "evala env (If_then_else b a1 a2) =
     (if evalb env b then evala env a1 else evala env a2)"
  "evala env (Sum a1 a2) = evala env a1 + evala env a2"
  "evala env (Diff a1 a2) = evala env a1 - evala env a2"
  "evala env (Var v) = env v"
  "evala env (Num n) = n"

  "evalb env (Less a1 a2) = (evala env a1 < evala env a2)"
  "evalb env (And b1 b2) = (evalb env b1 & evalb env b2)"
  "evalb env (Or b1 b2) = (evalb env b1 & evalb env b2)"
```

Since the value of an expression depends on the value of its variables, the functions `evala` and `evalb` take an additional parameter, an *environment* of type `'a => nat`, which maps variables to their values.

Similarly, we may define substitution functions `substa` and `substb` for expressions: The mapping `f` of type `'a => 'a aexp` given as a parameter is lifted canonically on the types `'a aexp` and `'a bexp`:

```
consts
  substa :: "['a => 'b aexp, 'a aexp] => 'b aexp"
  substb :: "['a => 'b aexp, 'a bexp] => 'b bexp"

primrec
  "substa f (If_then_else b a1 a2) =
     If_then_else (substb f b) (substa f a1) (substa f a2)"
  "substa f (Sum a1 a2) = Sum (substa f a1) (substa f a2)"
  "substa f (Diff a1 a2) = Diff (substa f a1) (substa f a2)"
  "substa f (Var v) = f v"
  "substa f (Num n) = Num n"

  "substb f (Less a1 a2) = Less (substa f a1) (substa f a2)"
  "substb f (And b1 b2) = And (substb f b1) (substb f b2)"
  "substb f (Or b1 b2) = Or (substb f b1) (substb f b2)"
```

In textbooks about semantics one often finds *substitution theorems*, which express the relationship between substitution and evaluation. For `'a aexp` and `'a bexp`, we can prove such a theorem by mutual induction, followed by simplification:

```
Goal
  "evala env (substa (Var(v := a')) a) =
     evala (env(v := evala env a')) a &
   evalb env (substb (Var(v := a')) b) =
     evalb (env(v := evala env a')) b";
by (induct_tac "a b" 1);
by (ALLGOALS Asm_full_simp_tac);
```

### Example: A substitution function for terms

Functions on datatypes with nested recursion, such as the type `term` mentioned in §2.6.1, are also defined by mutual primitive recursion. A substitution function `subst_term` on type `term`, similar to the functions `substa` and `substb` described above, can be defined as follows:

```
consts
  subst_term :: "['a => ('a,'b) term, ('a,'b) term] => ('a,'b) term"
  subst_term_list ::
    "['a => ('a,'b) term, ('a,'b) term list] => ('a,'b) term list"

primrec
  "subst_term f (Var a) = f a"
  "subst_term f (App b ts) = App b (subst_term_list f ts)"

  "subst_term_list f [] = []"
  "subst_term_list f (t # ts) =
     subst_term f t # subst_term_list f ts"
```

The recursion scheme follows the structure of the unfolded definition of type `term` shown in §2.6.1. To prove properties of this substitution function, mutual induction is needed:

```
Goal
  "(subst_term ((subst_term f1) o f2) t) =
     (subst_term f1 (subst_term f2 t)) &
   (subst_term_list ((subst_term f1) o f2) ts) =
     (subst_term_list f1 (subst_term_list f2 ts))";
by (induct_tac "t ts" 1);
by (ALLGOALS Asm_full_simp_tac);
```

**Example: A map function for infinitely branching trees**

Defining functions on infinitely branching datatypes by primitive recursion is just as easy. For example, we can define a function `map_tree` on `'a tree` as follows:

```
consts
  map_tree :: "('a => 'b) => 'a tree => 'b tree"

primrec
  "map_tree f (Atom a) = Atom (f a)"
  "map_tree f (Branch ts) = Branch (%x. map_tree f (ts x))"
```

Note that all occurrences of functions such as `ts` in the `primrec` clauses must be applied to an argument. In particular, `map_tree f o ts` is not allowed.

## 2.7.2 General recursive functions

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`.

```
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 $\tau$, then $\prec$ has to be a relation over $\tau$: 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 $\tau$ 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 Inductive and coinductive definitions

An **inductive definition** specifies the least set $R$ closed under given rules. (Applying a rule to elements of $R$ yields a result within $R$.) For example, a structural operational semantics is an inductive definition of an evaluation relation. Dually, a **coinductive definition** specifies the greatest set $R$

consistent with given rules. (Every element of $R$ can be seen as arising by applying a rule to elements of $R$.) An important example is using bisimulation relations to formalise equivalence of processes and infinite data structures.

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

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

This package is related to the ZF one, described in a separate paper,[6] which you should refer to in case of difficulties. The package is simpler than ZF's thanks to HOL's extra-logical automatic type-checking. The types of the (co)inductive sets determine the domain of the fixedpoint definition, and the package does not have to use inference rules for type-checking.

### 2.8.1 The result structure

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

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

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

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

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

`elims` is the list of elimination rule. This is for compatibility with ML scripts; within the theory the name is `cases`.

`elim` is the head of the list `elims`. This is for compatibility only.

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

[6]It appeared in CADE [14]; a longer version is distributed with Isabelle.

```
sig
val defs        : thm list
val mono        : thm
val unfold      : thm
val intrs       : thm list
val elims       : thm list
val elim        : thm
val mk_cases    : string -> thm
(Inductive definitions only)
val induct      : thm
(coinductive definitions only)
val coinduct    : thm
end
```

Figure 2.24: The ML result of a (co)inductive definition

For an inductive definition, the result structure contains the rule `induct`. For a coinductive definition, it contains the rule `coinduct`.

Figure 2.24 summarises the two result signatures, specifying the types of all these components.

## 2.8.2 The syntax of a (co)inductive definition

An inductive definition has the form

```
inductive     inductive sets
  intrs       introduction rules
  monos       monotonicity theorems
```

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

The `monos` section is optional; if present it is specified by a list of identifiers.

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

- The *constructor definitions* contain definitions of constants appearing in the introduction rules. In most cases it can be omitted.

### 2.8.3 *Monotonicity theorems

Each theory contains a default set of theorems that are used in monotonicity proofs. New rules can be added to this set via the `mono` attribute. Theory `Inductive` shows how this is done. In general, the following monotonicity theorems may be added:

- Theorems of the form $A \subseteq B \Longrightarrow M(A) \subseteq M(B)$, for proving monotonicity of inductive definitions whose introduction rules have premises involving terms such as $t \in M(R_i)$.
- Monotonicity theorems for logical operators, which are of the general form $[\![\cdots \to \cdots;\ \ldots;\ \cdots \to \cdots]\!] \Longrightarrow \cdots \to \cdots$. For example, in the case of the operator $\lor$, the corresponding theorem is

$$\frac{P_1 \to Q_1 \quad P_2 \to Q_2}{P_1 \lor P_2 \to Q_1 \lor Q_2}$$

- De Morgan style equations for reasoning about the "polarity" of expressions, e.g.

$$(\neg\neg P) \;=\; P \qquad\qquad (\neg(P \land Q)) \;=\; (\neg P \lor \neg Q)$$

- Equations for reducing complex operators to more primitive ones whose monotonicity can easily be proved, e.g.

$$(P \to Q) \;=\; (\neg P \lor Q) \qquad\qquad \texttt{Ball}\ A\ P \;\equiv\; \forall x\,.\ x \in A \to P\ x$$

### 2.8.4 Example of an inductive definition

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

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

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

For another example, here is a theory file defining the accessible part of a relation. The paper [14] discusses a ZF version of this example in more detail.

```
Acc = WF + Inductive +

consts acc :: "('a * 'a)set => 'a set"   (* accessible part *)

inductive "acc r"
  intrs
    accI "ALL y. (y, x) : r --> y : acc r ==> x : acc r"

end
```

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

## 2.9 Executable specifications

For validation purposes, it is often useful to *execute* specifications. In principle, specifications could be "executed" using Isabelle's inference kernel, i.e. by a combination of resolution and simplification. Unfortunately, this approach is rather inefficient. A more efficient way of executing specifications is to translate them into a functional programming language such as ML. Isabelle's built-in code generator supports this.

### 2.9.1 Invoking the code generator

The code generator is invoked via the `code_module` and `code_library` commands (see Fig. 2.25), which correspond to *incremental* and *modular* code generation, respectively.

**Modular** For each theory, an ML structure is generated, containing the code generated from the constants defined in this theory.

**Incremental** All the generated code is emitted into the same structure. This structure may import code from previously generated structures, which can be specified via `imports`. Moreover, the generated structure may also be referred to in later invocations of the code generator.

After the `code_module` and `code_library` keywords, the user may specify an optional list of "modes" in parentheses. These can be used to instruct the

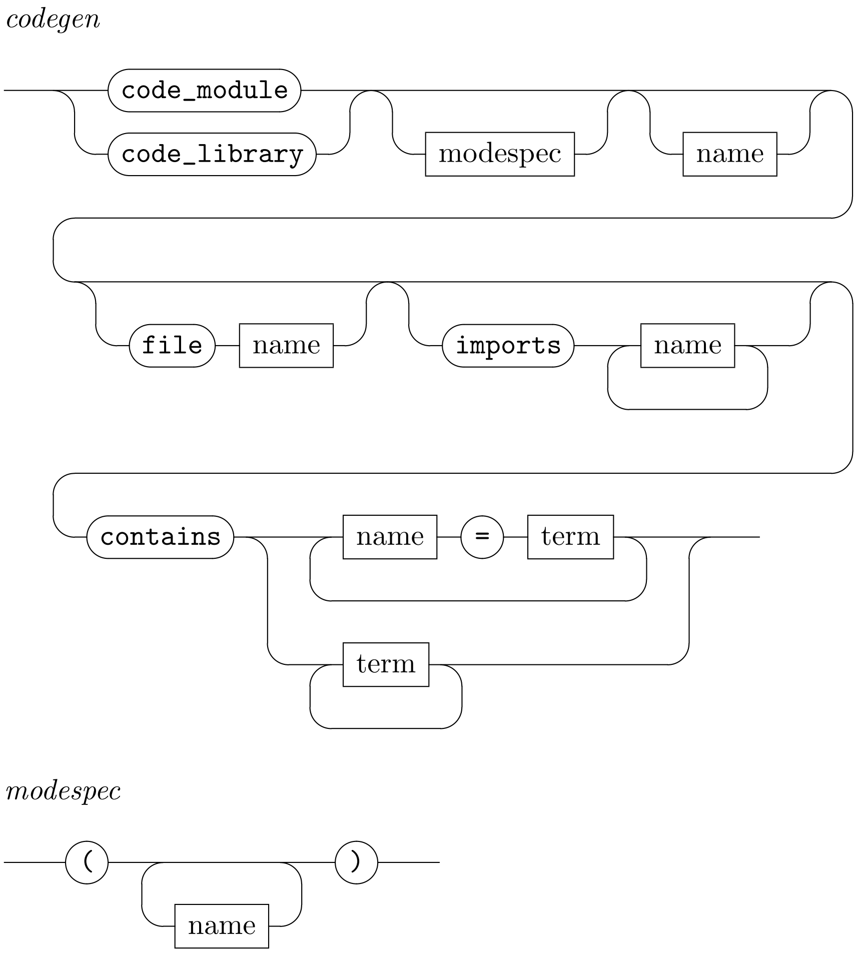

Figure 2.25: Code generator invocation syntax

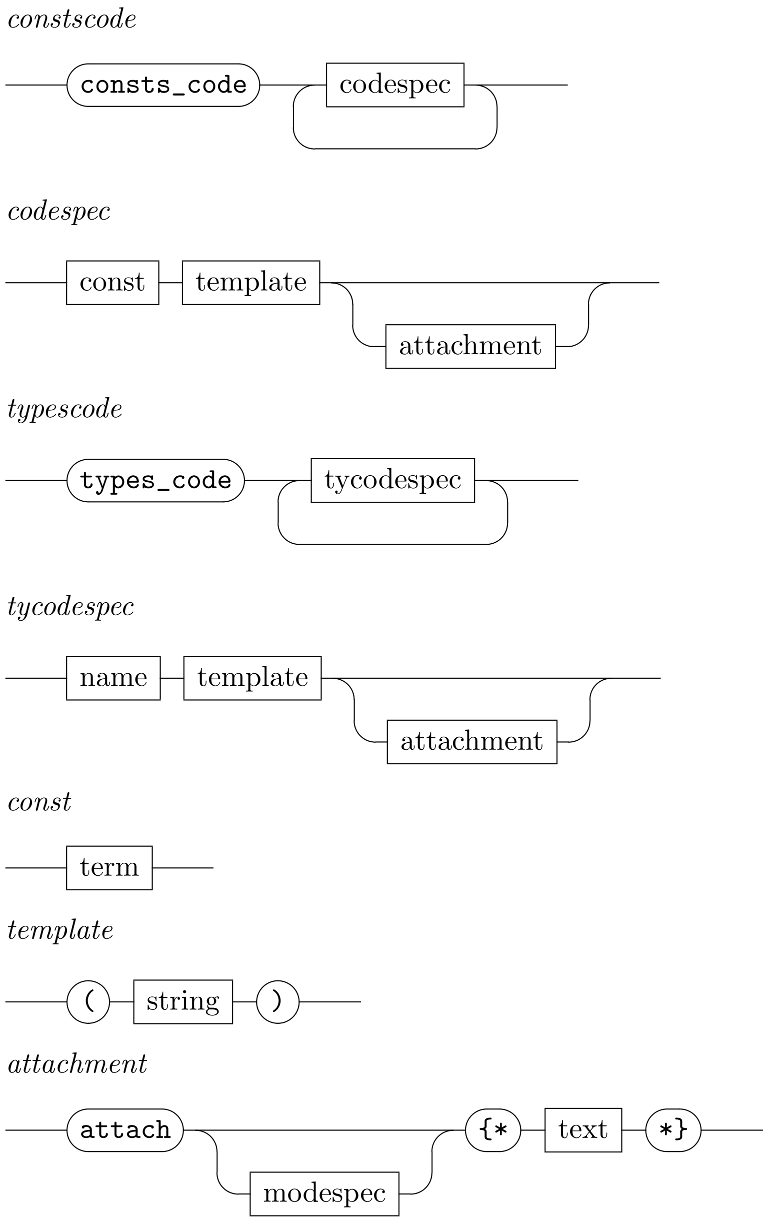

Figure 2.26: Code generator configuration syntax

code generator to emit additional code for special purposes, e.g. functions for converting elements of generated datatypes to Isabelle terms, or test data generators. The list of modes is followed by a module name. The module name is optional for modular code generation, but must be specified for incremental code generation. The code can either be written to a file, in which case a file name has to be specified after the `file` keyword, or be loaded directly into Isabelle's ML environment. In the latter case, the `ML` theory command can be used to inspect the results interactively. The terms from which to generate code can be specified after the `contains` keyword, either as a list of bindings, or just as a list of terms. In the latter case, the code generator just produces code for all constants and types occuring in the term, but does not bind the compiled terms to ML identifiers. For example,

```
code_module Test
contains
  test = "foldl op + (0::int) [1,2,3,4,5]"
```

binds the result of compiling the term `foldl op + (0::int) [1,2,3,4,5]` (i.e. `15`) to the ML identifier `Test.test`.

## 2.9.2 Configuring the code generator

When generating code for a complex term, the code generator recursively calls itself for all subterms. When it arrives at a constant, the default strategy of the code generator is to look up its definition and try to generate code for it. Constants which have no definitions that are immediately executable, may be associated with a piece of ML code manually using the `consts_code` command (see Fig. 2.26). It takes a list whose elements consist of a constant (given in usual term syntax – an explicit type constraint accounts for overloading), and a mixfix template describing the ML code. The latter is very much the same as the mixfix templates used when declaring new constants. The most notable difference is that terms may be included in the ML template using antiquotation brackets `{*` ... `*}`. A similar mechanism is available for types: `types_code` associates type constructors with specific ML code. For example, the declaration

```
types_code
  "*"     ("(_ */ _)")

consts_code
  "Pair"    ("(_,/ _)")
```

in theory `Product_Type` describes how the product type of Isabelle/HOL should be compiled to ML. Sometimes, the code associated with a constant or type may need to refer to auxiliary functions, which have to be emitted when the constant is used. Code for such auxiliary functions can be declared using `attach`. For example, the `wfrec` function from theory `Wellfounded_Recursion` is implemented as follows:

```
consts_code
  "wfrec"   ("\<module>wfrec?")
attach {*
fun wfrec f x = f (wfrec f) x;
*}
```

If the code containing a call to `wfrec` resides in an ML structure different from the one containing the function definition attached to `wfrec`, the name of the ML structure (followed by a "`.`") is inserted in place of "`\<module>`" in the above template. The "`?`" means that the code generator should ignore the first argument of `wfrec`, i.e. the termination relation, which is usually not executable.

Another possibility of configuring the code generator is to register theorems to be used for code generation. Theorems can be registered via the `code` attribute. It takes an optional name as an argument, which indicates the format of the theorem. Currently supported formats are equations (this is the default when no name is specified) and horn clauses (this is indicated by the name `ind`). The left-hand sides of equations may only contain constructors and distinct variables, whereas horn clauses must have the same format as introduction rules of inductive definitions. For example, the declaration

```
lemma Suc_less_eq [iff, code]: "(Suc m < Suc n) = (m < n)" ⟨...⟩
lemma [code]: "((n::nat) < 0) = False" by simp
lemma [code]: "(0 < Suc n) = True" by simp
```

in theory `Nat` specifies three equations from which to generate code for `<` on natural numbers.

## 2.9.3 Specific HOL code generators

The basic code generator framework offered by Isabelle/Pure has already been extended with additional code generators for specific HOL constructs. These include datatypes, recursive functions and inductive relations. The code generator for inductive relations can handle expressions of the form $(t_1, \ldots, t_n) \in r$, where $r$ is an inductively defined relation. If at least one of the $t_i$ is a dummy pattern "_", the above expression evaluates to a sequence of possible answers. If all of the $t_i$ are proper terms, the expression evaluates to a boolean value.

```
theory Test = Lambda:
```

```
code_module Test
contains
  test1 = "Abs (Var 0) ∘ Var 0 -> Var 0"
  test2 = "Abs (Abs (Var 0 ∘ Var 0) ∘ (Abs (Var 0) ∘ Var 0)) -> _"
```

In the above example, `Test.test1` evaluates to the boolean value `true`, whereas `Test.test2` is a sequence whose elements can be inspected using `Seq.pull` or similar functions.

```
ML {* Seq.pull Test.test2 *}  -- {* This is the first answer *}
ML {* Seq.pull (snd (the it)) *}  -- {* This is the second answer *}
```

The theory underlying the HOL code generator is described more detailed in [2]. More examples that illustrate the usage of the code generator can be found e.g. in theories `MicroJava/J/JListExample` and `MicroJava/JVM/JVMListExample`.

## 2.10 The examples directories

Directory `HOL/Auth` contains theories for proving the correctness of cryptographic protocols [17]. The approach is based upon operational semantics rather than the more usual belief logics. On the same directory are proofs for some standard examples, such as the Needham-Schroeder public-key authentication protocol and the Otway-Rees protocol.

Directory `HOL/IMP` contains a formalization of various denotational, operational and axiomatic semantics of a simple while-language, the necessary equivalence proofs, soundness and completeness of the Hoare rules with respect to the denotational semantics, and soundness and completeness of a verification condition generator. Much of development is taken from Winskel [22]. For details see [12].

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

Directory `HOL/MiniML` contains a formalization of the type system of the core functional language Mini-ML and a correctness proof for its type inference algorithm $\mathcal{W}$ [8, 10].

Directory `HOL/Lambda` contains a formalization of untyped $\lambda$-calculus in de Bruijn notation and Church-Rosser proofs for $\beta$ and $\eta$ reduction [11].

Directory `HOL/Subst` contains Martin Coen's mechanization of a theory of substitutions and unifiers. It is based on Paulson's previous mechanisation in LCF [13] of Manna and Waldinger's theory [7]. It demonstrates a complicated use of `recdef`, with nested recursion.

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

- Theory `PropLog` proves the soundness and completeness of classical propositional logic, given a truth table semantics. The only connective is $\to$. A Hilbert-style axiom system is specified, and its set of theorems defined inductively. A similar proof in ZF is described elsewhere [15].
- Theory `Term` defines the datatype `term`.
- Theory `ABexp` defines arithmetic and boolean expressions as mutually recursive datatypes.
- The definition of lazy lists demonstrates methods for handling infinite data structures and coinduction in higher-order logic [16].[7] Theory `LList` defines an operator for corecursion on lazy lists, which is used to define a few simple functions such as map and append. A coinduction principle is defined for proving equations on lazy lists.
- Theory `LFilter` defines the filter functional for lazy lists. This functional is notoriously difficult to define because finding the next element meeting the predicate requires possibly unlimited search. It is not computable, but can be expressed using a combination of induction and corecursion.
- Theory `Exp` illustrates the use of iterated inductive definitions to express a programming language semantics that appears to require mutual induction. Iterated induction allows greater modularity.

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

- Theory `Recdef` presents many examples of using `recdef` to define recursive functions. Another example is `Fib`, which defines the Fibonacci function.
- Theory `Primes` defines the Greatest Common Divisor of two natural numbers and proves a key lemma of the Fundamental Theorem of Arithmetic: if $p$ is prime and $p$ divides $m \times n$ then $p$ divides $m$ or $p$ divides $n$.
- Theory `Primrec` develops some computation theory. It inductively defines the set of primitive recursive functions and presents a proof that Ackermann's function is not primitive recursive.

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

- File `cla.ML` demonstrates the classical reasoner on over sixty predicate calculus theorems, ranging from simple tautologies to moderately difficult problems involving equality and quantifiers.

- File `meson.ML` contains an experimental implementation of the MESON proof procedure, inspired by Plaisted [20]. It is much more powerful than Isabelle's classical reasoner. But it is less useful in practice because it works only for pure logic; it does not accept derived rules for the set theory primitives, for example.

- File `mesontest.ML` contains test data for the MESON proof procedure. These are mostly taken from Pelletier [19].

- File `set.ML` proves Cantor's Theorem, which is presented in §2.11 below, and the Schröder-Bernstein Theorem.

- Theory `MT` contains Jacob Frost's formalization [5] of Milner and Tofte's coinduction example [9]. This substantial proof concerns the soundness of a type system for a simple functional language. The semantics of recursion is given by a cyclic environment, which makes a coinductive argument appropriate.

## 2.11 Example: Cantor's Theorem

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

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

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

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

```
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] Peter Andrews. *An Introduction to Mathematical Logic and Type Theory: to Truth through Proof.* Computer Science and Applied Mathematics. Academic Press, 1986.

[2] Stefan Berghofer and Tobias Nipkow. Executing higher order logic. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *Types for Proofs and Programs: TYPES'2000*, volume 2277 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[3] 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.

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

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

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

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

[8] Robin Milner. A theory of type polymorphism in programming. *J. Comp. Sys. Sci.*, 17:348–375, 1978.

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

[10] Wolfgang Naraschewski and Tobias Nipkow. Type inference verified: Algorithm W in Isabelle/HOL. In E. Giménez and C. Paulin-Mohring,

editors, *Types for Proofs and Programs: Intl. Workshop TYPES '96*, volume 1512 of *Lecture Notes in Computer Science*, pages 317–332. Springer-Verlag, 1998.

[11] Tobias Nipkow. More Church-Rosser proofs (in Isabelle/HOL). In M. McRobbie and J.K. Slaney, editors, *Automated Deduction — CADE-13*, volume 1104 of *Lecture Notes in Computer Science*, pages 733–747. Springer-Verlag, 1996.

[12] Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10:171–186, 1998.

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

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

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

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

[17] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

[18] 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.

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

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

[21] 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.

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

# Index