Revised version for CSE580 **January 12, 2001**

HOL

A Machine Oriented Formulation of Higher Order Logic

Mike Gordon

Computer Laboratory

Pembroke Street

New Museums Site

Cambridge CB2 3QG

HOL is a computer system for generating proofs in a version of higher order logic derived from Church's simple theory of types.

This paper is the original description of the logic underlying the HOL system. Since it was written the system has changed enormously, but the logic has been relatively stable. The main changes have been in the organisation of theories and some enhancements to the syntactic resources of the logic (e.g. records and user-defined mixfix syntax are now supported).

A more modern and technically sophisticated description (written largely by Dr. Andrew Pitts) can be found as Part I of "The HOL System Description" at

`http://www.cl.cam.ac.uk/Research/HVG/FTP/`

The HOL documentation is currently being actively updated for HOL98 but still contains some obsolete material. However, it, and the Tutorial (also available at the above website), are the best that is available.

Contents

1 Introduction

In this paper we describe a formal language intended as a basis for hardware specification and verification. This language is not new; the only originality in what follows lies in the presentation of details. Considerable effort has gone into making the formalism suitable for manipulation by computer (e.g. it has a type system for which there is a powerful type checking algorithm [Milner (78)]).

Any language intended for hardware specification and verification must be capable of representing the mathematics needed in reasoning about digital devices. This includes theories of bits, bitstrings, numbers, pairs, lists, functions, Fourier transforms etc., as well as the specialized theories of time that underlie formalisms such as Interval Temporal Logic [Halpern *et al.*].

Mathematics is usually formalized in Set Theory, but for our purposes Higher Order Logic is more appropriate. This is because the style of hardware specification we want to support makes extensive use of higher order functions.

The logic described here underlies an automated proof generator called HOL. This acronym will be used both for the computer system and for the logic embedded in it. If disambiguation between these is needed I will call the former the "HOL system" and the latter the "HOL logic".

Various other projects to automate Higher Order Logic are in progress. These include the TPS theorem prover being developed at Carnegie-Mellon University [Andrews *et al.*] and the EKL proof checker at Stanford [Ketonen & Weening]. The idea of using Higher Order Logic for hardware specification and verification is due to Keith Hanna of the University of Kent [Hanna & Daeche].

The HOL logic is a version of Church's simple theory of types [Church] modified by:

- allowing types to contain variables (i.e. be *polymorphic*) and
- simplifying the axioms and rules of inference.

The exact syntax of the logic is defined relative to a *theory*, which determines the types and constants that are available. Theories are developed incrementally starting from the standard theories `bool` (of truth-values or booleans) and `ind` (of individuals). Mechanisms are provided by the HOL system for setting up new theories.

2 Overview of Higher Order Logic

The HOL logic uses standard predicate calculus notation, for example:

- "$P(x)$" means "x has property P",
- "$\neg t$" means "not t",
- "$t_1 \wedge t_2$" means "t_1 and t_2",
- "$t_1 \vee t_2$" means "t_1 or t_2",
- "$t_1 \Rightarrow t_2$" means "t_1 implies t_2",
- "$\forall x.\ t[x]$" means "for all x it is the case that $t[x]$",
- "$\exists x.\ t[x]$" means "for some x it is the case that $t[x]$",
- "$\exists! x.\ t[x]$" means "there is a unique x such that $t[x]$".

Here t, t_1 and t_2 stand for arbitrary *terms*, and $t[x]$ stands for some term containing the variable x.

The HOL logic uses four kinds of terms. These will be explained in detail later, but here is a quick overview:

1. **Variables.** These are sequences of letters or digits beginning with a letter. For example: x, y, P, $This\_is\_a\_single\_variable$. Certain other strings are allowed also (e.g. $I'm\_a\_variable$).

2. **Constants.** These have the same syntax as variables, but stand for fixed values. Whether an identifier is a variable or a constant is determined by a theory, this will be explained later. Examples of constants are: T, F (with respect to the theory `bool` of truth-values), 0, 1, 2, ... (with respect to the theory `num` of numbers), + (with respect to the theory `arithmetic` of arithmetic).

3. **Function applications.** These have the general form $t_1\ t_2$ where t_1 and t_2 are terms, an example is $P\ 0$. Brackets can be inserted around terms to increase readability or to enforce grouping, thus $P\ 0$ is equivalent to $P(0)$. Binary function constants can be declared (with respect to a theory) to be infixed. This provides a mechanism enabling one to write $t_1 + t_2$ instead of $+\ t_1\ t_2$.

4. **Lambda-terms.** These denote functions and have the form $\lambda x.\ t$ (where x is a variable and t a term). For example, $\lambda n.\ n + 1$ denotes the successor function.

HOL provides some syntactic mechanisms to support conventional logical and mathematical notations. For example, if one declares the constants $\Rightarrow$ and $+$ to be *infixes* and the constant $\forall$ to be a *binder* then $\forall n.\ P(n) \Rightarrow P(n+1)$ is written instead of $\forall(\lambda n. \Rightarrow(P(n))(P(+\ n\ 1)))$.

Higher Order Logic generalizes First Order Logic by allowing *higher order* variables — i.e. variables ranging over functions and predicates. For example, the induction axiom for natural numbers can be written as:

$$\forall P.\ P(0) \wedge (\forall n.\ P(n) \Rightarrow P(n+1)) \Rightarrow \forall n.\ P(n)$$

and the legitimacy of simple recursive definitions (the Peano-Lawvere Axiom [MacLane and Birkhoff]) can be expressed by:

$$\forall n_0.\ \forall f.\ \exists! s.\ (s(0) = n_0) \wedge (\forall n.\ s(n+1) = f(s(n)))$$

Sentences like these are not allowed in first order logic: in the first example above P ranges over predicates; in the second example f and s range over functions.

3 Terms

The four kinds of terms in the HOL logic are *variables*, *constants*, *applications* (of a function to an argument) and *abstractions* (also called λ-terms). These are described in detail below.

3.1 Variables and constants

Variables and constants stand for values. They can be any sequence of letters, digits, primes ($'$) or underlines (_) starting with a letter. In addition there are some special symbols for the logical operators including: the equals sign ($=$), the equivalence symbol ($\equiv$), the negation symbol ($\neg$), the conjunction symbol ($\wedge$), the disjunction symbol ($\vee$), the implication symbol ($\Rightarrow$), the universal quantifier ($\forall$), the existential quantifier ($\exists$), the unique existence quantifier ($\exists!$) and Hilbert's epsilon symbol (ε). Other allowed variable or constant symbols are the pairing symbol (comma: ,), the numerals 0, 1, 2 etc., the arithmetic functions $+$, $-$, $\times$ and $/$, and the arithmetic relations $<$, $>$, $\leq$ and $\geq$.

Whether an identifier is a variable or a constant is determined by a theory. For example, $\wedge$ is a constant of the theory `bool`, and $+$ is a constant of the theory `num`. One can thus only parse a term relative to a theory. We will use the convention that $\mathsf{sans\ serif}$ identifiers and non-alphabetical symbols are constants, and *italic* identifiers are variables. Arbitrary terms will usually be denoted by t, t_1, t_2, etc.

3.2 Function applications

Terms of the form $t_1(t_2)$ are called *applications* or *combinations*. The subterm t_1 is called the *operator* (or *rator*) and the term t_2 is called the *operand* (or *rand* or *argument*). The result of such a function application can itself be a function and thus terms like $(t_1(t_2))(t_3)$ are allowed. Functions that take functions as arguments or return functions as results are called *higher order*.

To save writing brackets, function applications can be written as $f\ x$ instead of $f(x)$. More generally we adopt the usual convention that $t_1\ t_2\ t_3\ \cdots\ t_n$ abbreviates $(\ \cdots\ ((t_1\ t_2)\ t_3)\ \cdots\ t_n)$ i.e. application associates to the left.

3.3 Lambda-terms

HOL provides *lambda-terms* (also called *λ-terms* or *abstractions*) for denoting functions. Such a term has the form $\lambda x.\ t$ (where t is a term) and denotes the function f defined by:

$$f(x) \ = \ t$$

For example, $\lambda n.\ \mathsf{cos}(\mathsf{sin}(n))$ denotes the function f such that:

$$f(n) \ = \ \mathsf{cos}(\mathsf{sin}(n))$$

thus: $f(1) = \mathsf{cos}(\mathsf{sin}(1))$, $f(2) = \mathsf{cos}(\mathsf{sin}(2))$ etc. The variable x and term t are called respectively the *bound variable* and *body* of the λ-expression $\lambda x.\ t$. An occurrence of the bound variable in the body is called a *bound* occurrence. If an occurrence is not bound it is called *free*.

4 Types

The increased expressive power gained by allowing higher order variables is dangerous. Consider the predicate P defined by:

$$\mathsf{P}\ x \;=\; \neg(x\ x)$$

from this definition it follows that:

$$\mathsf{P}\ \mathsf{P} \;=\; \neg(\mathsf{P}\ \mathsf{P})$$

which is a version of Russell's paradox. Russell invented a method for preventing such inconsistencies based on the use of *types* [Hatcher]. HOL uses a simplification of Russell's type system due to Church [Church] with extensions developed by Milner [Milner (78)].

Types are expressions that denote sets of values, they are either *atomic* or *compound*. Examples of atomic types are:

$$\mathit{bool}, \qquad \mathit{ind}, \qquad \mathit{num}, \qquad \mathit{real}$$

these denote the sets of booleans, individuals, natural numbers and real numbers respectively. Compound types are built from atomic types (or other compound types) using *type operators*. For example, if σ, σ_1 and σ_2 are types then so are:

$$\sigma\ \mathit{list}, \qquad \sigma_1{\rightarrow}\sigma_2$$

where *list* is a unary type operator and $\rightarrow$ is an infixed binary type operator. The type σ *list* denotes the set of lists of values of type σ and the type $\sigma_1{\rightarrow}\sigma_2$ denotes the set of functions with domain denoted by σ_1 and range denoted by σ_2. In general compound types are expressions of the form:

$$(\sigma_1,\ \ldots\ ,\sigma_n)\mathit{op}$$

where *op* is a type operator and $\sigma_1,\ \ldots\ ,\ \sigma_n$ are types. If the operator has only one argument then the brackets can be omitted (hence σ *list*); the type $\sigma_1{\rightarrow}\sigma_2$ is an *ad hoc* abbreviation for $(\sigma_1, \sigma_2)\mathit{fun}$. We will use lower case *slanted* identifiers for particular types, and greek letters (mostly σ) to range over arbitrary types.

We require each variable and constant occurring in a HOL term to be assigned a type. Variables with the same name but different types are regarded as different. We indicate that x has type σ by writing $x{:}\sigma$. Thus $x{:}\sigma_1$ is a different variable from

$x{:}\sigma_2$ if and only if σ_1 and σ_2 are different. Starting from the types of the variables and constants in a term the rules **[Ty1]** and **[Ty2]** below determine whether the term is *well-typed* and if it is what its type is. If t is a well-typed term with type σ we write $t{:}\sigma$. Only well typed terms are allowed in HOL. This restriction ensures that each term is meaningful (if $t{:}\sigma$ then t denotes a member of the set denoted by σ) and is sufficient to block the derivation of Russell's paradox.

[Ty1] A term of the form $t_1\ t_2$ is well-typed with type σ if and only if for some type σ'

1. t_1 is well-typed with type $\sigma'{\rightarrow}\sigma$, and
2. t_2 is well-typed with type σ'.

[Ty2] A term of the form $\lambda x.\ t$ is well-typed with type σ if and only if σ has the form $\sigma_1{\rightarrow}\sigma_2$ and:

1. x has type σ_1, and
2. t is a well-typed term with type σ_2.

In some formulations of higher-order logic the types of variables have to be written down explicitly. For example, one would not be allowed to write $\lambda x.\ \mathsf{cos}(\mathsf{sin}(x))$ but instead one would have to write $\lambda x{:}\mathit{real}.\ \mathsf{cos}(\mathsf{sin}(x{:}\mathit{real}))$. In HOL we allow the types of variables to be omitted if they can be inferred from the context (using the declared types of the constants). The type inference algorithm used by the HOL system is due to Robin Milner [Milner(78)]. In the absence of explicit type information this algorithm makes the assumption that variables with the same name have the same type and it would thus infer that both occurrences of x in $\lambda x.\ cos(sin(x))$ have type *real*. To get the term $\lambda x{:}\mathit{bool}.\ \mathsf{sin}(\mathsf{cos}(x{:}\mathit{real}))$ (which is a well-typed term of type $\mathit{bool}{\rightarrow}\mathit{real}$ that denotes a constant function) one must write the types in explicitly. Note that in this term the bound variable has type *bool* and is thus different from the other occurrence of x with type *real* (which is thus a free occurrence).

Consider the term $(x\ x)$ that was used in formulating Russell's Paradox. This has the form $(t_1\ t_2)$ with $t_1 = x$ and $t_2 = x$. Thus if $(x\ x)$ is to be well-typed then for some types σ and σ' the first occurrence of the variable x must have type $\sigma'{\rightarrow}\sigma$ and the second occurrence type σ'. Thus if the equation $\mathsf{P}\ x\ =\ \neg(x\ x)$ is to be well-typed then the x to the left of the $=$ must be different from at least one of the two xs in the right of it (since these two xs have different types). In HOL it is only valid to instantiate a variable with a term if the term has the same type as the variable. It follows that one cannot derive the paradoxical $\mathsf{P}\ \mathsf{P} \ = \ \neg(\mathsf{P}\ \mathsf{P})$ by instantiating x to P in $\mathsf{P}\ x \ = \ \neg(x\ x)$ because whatever type P has it must be different from the type of at least one of the xs to the right of the $=$. Russell's paradox is thus avoided.

HOL adopts the usual convention that $\sigma_1{\rightarrow}\sigma_2{\rightarrow}\sigma_3{\rightarrow}\ \cdots\ \sigma_n{\rightarrow}\sigma$ is an abbreviation for $\sigma_1{\rightarrow}(\sigma_2{\rightarrow}(\sigma_3{\rightarrow}\ \cdots\ (\sigma_n{\rightarrow}\sigma)\ \cdots))$ i.e. $\rightarrow$ associates to the right. This convention blends well with the left associativity of function application because if f has type $\sigma_1{\rightarrow}\ \cdots\ \sigma_n{\rightarrow}\sigma$ and $t_1, \ldots, t_n$ have types $\sigma_1, \ldots, \sigma_n$ respectively then $f\ t_1\ \cdots\ t_n$ is a well-typed term of type σ.

The notation $\lambda x_1\ x_2\ \cdots\ x_n.\ t$ abbreviates $\lambda x_1.\ \lambda x_2.\ \ \cdots\ \lambda x_n.\ t$. The scope of the "." after a λ extends as far to the right as possible. Thus, for example, $\lambda b.\ b = \lambda x.\ \mathsf{T}$ means $\lambda b.\ (b = (\lambda x.\ \mathsf{T}))$ not $(\lambda b.\ b) = (\lambda x.\ \mathsf{T})$.

4.1 Type variables and polymorphism

Consider the function twice defined by:

$$\mathsf{twice} \ = \ \lambda f.\ \lambda x.\ f(f(x))$$

If f is a function then $\mathsf{twice}(f)$, the result of applying twice to f, is the function $\lambda x.\ f(f(x))$; twice is thus a function-returning function, i.e. it is higher order. For example, if sin is a trigonometric function with type *real*$\rightarrow$*real*, then $\mathsf{twice}(\mathsf{sin})$ is $\lambda x.\ \mathsf{sin}(\mathsf{sin}(x))$ which is the function taking the sin of the sin of its argument, a function of type *real*$\rightarrow$*real*, and if not is a boolean function with type *bool*$\rightarrow$*bool*, then $\mathsf{twice}(\mathsf{not})$ is $\lambda x.\ \mathsf{not}(\mathsf{not}(x))$ which is the function taking the double negation of its argument, a function of type *bool*$\rightarrow$*bool*.

What then is the type of the function twice? Since $\mathsf{twice}(\mathsf{sin})$ has type *real*$\rightarrow$*real* it would appear that twice has the type $(\mathit{real}{\rightarrow}\mathit{real}){\rightarrow}(\mathit{real}{\rightarrow}\mathit{real})$. However, since $\mathsf{twice}(\mathsf{not})$ has type *bool*$\rightarrow$*bool* it would also appear that twice has the type $(\mathit{bool}{\rightarrow}\mathit{bool}){\rightarrow}(\mathit{bool}{\rightarrow}\mathit{bool}$ Thus twice would appear to have two different types. In Church's Simple Type Theory this would not be allowed and we would have to define two functions, $\mathsf{twice}_{(\mathit{real}\rightarrow\mathit{real})\rightarrow(\mathit{real}\rightarrow\mathit{real})}$ and $\mathsf{twice}_{(\mathit{bool}\rightarrow\mathit{bool})\rightarrow(\mathit{bool}\rightarrow\mathit{bool})}$ say. In HOL, *type variables* are used to overcome this messiness; for example, if α is a type variable then twice can be given the type $(\alpha{\rightarrow}\alpha){\rightarrow}(\alpha{\rightarrow}\alpha)$ and then it behaves as though it has all instances of this that can be obtained by replacing α by a type. Types containing type variables are called *polymorphic*, ones not containing variables are *monomorphic*. We shall call a term polymorphic or monomorphic if its type is polymorphic or monomorphic respectively. We will use α, β, γ etc. for type variables.

An *instance* of a type σ is a type obtained by replacing zero or more type variables in σ by types. Here are some instances of $(\alpha{\to}\alpha){\to}(\alpha{\to}\alpha)$:

$$(\mathit{real}{\to}\mathit{real}){\to}(\mathit{real}{\to}\mathit{real})$$
$$(\mathit{bool}{\to}\mathit{bool}){\to}(\mathit{bool}{\to}\mathit{bool})$$
$$((\alpha{\to}\mathit{bool}){\to}(\alpha{\to}\mathit{bool})){\to}((\alpha{\to}\mathit{bool}){\to}(\alpha{\to}\mathit{bool}))$$

In these examples α has been replaced by real, bool and $\alpha{\to}\mathit{bool}$ respectively. The only instances of monomorphic types are themselves.

When constants are declared (a process that will be explained when we describe theories) they must be given a type. If this type is polymorphic then for the purposes of type checking the constant behaves as though it is assigned every instance of the type. For example, if twice were declared as a constant with type $(\alpha{\to}\alpha){\to}(\alpha{\to}\alpha)$, then the terms $\mathsf{twice}(\mathsf{sin})$ and $\mathsf{twice}(\mathsf{not})$ would be well-typed.

5 Special Syntactic Forms

Certain applications are conventionally written in special ways, for example:

- $+\ t_1\ t_2$ is written $t_1 + t_2$
- $,\ t_1\ t_2$ is written (t_1, t_2)
- $\forall(\lambda x.\ t)$ is written $\forall x.\ t$

The HOL logic enables constants to be given a special syntactic status (relative to a theory) to support such forms. For example, + and , are examples of *infixes* and $\forall$ is an example of a *binder*. Some other *ad hoc* syntactic forms are also allowed, these are explained below.

5.1 Infixes

Constants with types of the form $\sigma_1{\to}\sigma_2{\to}\sigma_3$ can be declared as *infixes*. If f is an infixed constant then applications are written as $t_1\ \mathsf{f}\ t_2$ rather than as $\mathsf{f}\ t_1\ t_2$.

Standard examples of infixes are the arithmetic functions $+$, $\times$ etc.The infix status of a constant can be suppressed by preceding it with "\$". Thus $\$+\ m\ n$ is equivalent to $m+n$. Whether a constant is an infix or not has no logical significance, it is merely syntactic. The parser of the HOL system translates terms of the form t_1 f t_2 into the same internal representation as terms of the form \$f $t_1\ t_2$.

Examples of infixes are the following constants of the theory `bool`:

$$\begin{array}{ll} \wedge\ :\ bool{\rightarrow}bool{\rightarrow}bool & \text{(Conjunction - i.e. ``and'')} \\ \vee\ :\ bool{\rightarrow}bool{\rightarrow}bool & \text{(Disjunction - i.e. ``or'')} \\ \Rightarrow\ :\ bool{\rightarrow}bool{\rightarrow}bool & \text{(Implication - i.e. ``implies'')} \\ \equiv\ :\ bool{\rightarrow}bool{\rightarrow}bool & \text{(Equivalence - i.e. ``if and only if'')} \end{array}$$

Equality is also an infixed constant; it is polymorphic:

$$= : \alpha{\rightarrow}\alpha{\rightarrow}bool$$

Equivalence ($\equiv$) is equality ($=$) restricted to booleans. The constants $\wedge$, $\vee$, $\Rightarrow$, $\equiv$ and $=$ are all infixes. The only primitive propositional constants are $=$ and $\Rightarrow$, the others can all be defined in terms of these. This is explained below in the section on the theory `bool`.

5.2 Binders

It is sometimes more readable to write $\mathsf{f}\ x.\ t$ instead of $\mathsf{f}(\lambda x.\ t)$. For example, in HOL the quantifiers $\forall$ and $\exists$ are polymorphic constants:

$$\begin{array}{l} \forall : (\alpha{\rightarrow}bool){\rightarrow}bool \\ \exists : (\alpha{\rightarrow}bool){\rightarrow}bool \end{array}$$

The idea is that if $P : \sigma{\rightarrow}bool$, then $\forall(P)$ is true if $P(x)$ is true for all x and $\exists(P)$ is true if $P(x)$ is true for some x. Instead of writing $\forall(\lambda x.\ t)$ and $\exists(\lambda x.\ t)$ it is nice to be able to use the more conventional forms $\forall x.\ t$ and $\exists x.\ t$.

Any constant f with a type of the form $(\sigma_1{\rightarrow}\sigma_2){\rightarrow}\sigma_3$ can be declared to be a *binder*. If this is done then instead of writing:

$$\mathsf{f}(\lambda x_1.\ \mathsf{f}(\lambda x_2.\ \cdots\ \mathsf{f}(\lambda x_n.\ t)\ \cdots))$$

one can write:

$$\mathsf{f}\ x_1\ \cdots\ x_n.\ t$$

As with infixes, the binder status of a constant is purely syntactic and can be suppressed with "\$".

Recall the statement of mathematical induction:

$$\forall P.\ P(0) \wedge (\forall n.\ P(n) \Rightarrow P(n+1)) \Rightarrow \forall n.\ P(n)$$

This is a term of HOL of type *bool*; it is the same as the unreadable:

$$\$\forall(\lambda P.\ \$\Rightarrow(\$\wedge(P\ 0)(\$\forall(\lambda n.\ \$\Rightarrow(P\ n)(P(\$+\ \ n\ 1)))))(\$\forall(\lambda n.\ Pn)))$$

The quantifiers $\forall$ and $\exists$ are not primitive in HOL. In the section on the theory `bool` we explain how they can be defined. The existential quantifier is defined in terms of Hilbert's ε-operator which is described later.

5.3 Pairs and tuples

A function of n arguments can be represented as a higher order function of 1 argument that returns a function of n-1 arguments. Thus $\lambda m.\ \lambda n.\ m^2 + n^2$ represents the 2 argument function that sums the squares of its arguments. Functions of this form are called *curried*. An alternative way of representing multiple argument functions is as single argument functions taking *tuples* as arguments. To handle tuples HOL has a binary type operator *prod*. If $t_1{:}\sigma_1$ and $t_2{:}\sigma_2$ then the term (t_1, t_2) has type $(\sigma_1, \sigma_2)prod$ and denotes the pair of values. The type $(\sigma_1, \sigma_2)prod$ can also be written as $\sigma_1{\times}\sigma_2$. Another representation of the sum-squares function would be as a constant, sumsq say, of type $(num{\times}num){\to}num$ defined by:

$$\mathsf{sumsq}(m, n) \ = \ m^2 + n^2$$

A term of the form (t_1, t_2) is equivalent to the term $\$,\ t_1\ t_2$ where "," is a polymorphic infixed constant of type $\alpha{\to}\beta{\to}(\alpha{\times}\beta)$. Instead of having tuples as primitive HOL (following LCF) treats them as iterated pairs. Thus the term:

$$(t_1, t_2,\ \ldots\ , t_{n-1}, t_n)$$

is an abbreviation for:

$$(t_1, (t_2,\ \ldots\ , (t_{n-1}, t_n)\ \ldots\ ))$$

i.e. "," associates to the right. To match this, the infixed type operator $\times$ also associates to the right so that if $t_1{:}\sigma_1$, $\ldots$, $t_n{:}\sigma_n$ then:

$$(t_1,\ \ldots\ , t_n) : \sigma_1{\times}\ \cdots\ {\times}\sigma_n$$

The type operator *prod* can be defined in terms of *fun* and thus pairing need not be primitive. We show how to do this in Appendix B.

5.4 Lists

The theory `list` (see Appendix B) introduces types σ *list*, together with constants Nil and Cons of types α *list* and $\alpha{\rightarrow}(\alpha\ \mathit{list}){\rightarrow}(\alpha\ \mathit{list})$ respectively. A term with type σ *list* denotes a list of values all of type σ. Nil is the empty list; HOL allows $[\ ]$ as an alternative form of Nil and $[t_1;\cdots;t_n]$ as an alternative form for $\mathsf{Cons}\ t_1(\mathsf{Cons}\ t_2\cdots(\mathsf{Cons}\ t_n\ \mathsf{Nil})\cdots))$.

The difference between lists and tuples is:

1. different lists of a given type can contain different numbers of elements, but all tuples of a given type contain exactly the same numbers of elements;
2. the elements of a list must all have the same type but elements of tuples can have different types.

5.5 Conditionals

The theory `bool` contains a constant Cond which is defined so that $\mathsf{Cond}\ t\ t_1\ t_2$ means "if t then t_1 else t_2". The special syntax $(t \to t_1 \mid t_2)$ is provided for such terms. The original conditional notation due to McCarthy used "," instead of "|".

6 Formulae, sequents, axioms and theorems

Unlike first order logic HOL has no separate syntactic class of *formulae*, their role is played by boolean terms (i.e. terms of type *bool*).

A *sequent* (Γ, t) consists of a finite set of boolean terms Γ called the *assumptions* together with a boolean term t called the *conclusion*. Think of (Γ, t) as asserting that "if every term in Γ is equivalent to T then so is t".

A *theorem* is a sequent that is either an *axiom* or follows from theorems by a *rule of inference*. Axioms are sequents that are just postulated to be theorems; rules of inference are procedures for deducing new theorems from existing ones. If (Γ, t) is a theorem we write $\Gamma\ \vdash\ t$, if Γ is empty we write $\vdash\ t$.

6.1 Definitions

Definitions are axioms of the form $\vdash\ \mathsf{c} = t$ where c is a new constant and t is a closed term (i.e. a term without any free variables) that doesn't contain c. Such a definition just introduces the constant c as an abbreviation for the term t. The requirement that c may not occur in t prevents definitions from being recursive, this is to rule out inconsistent 'definitions' like $\vdash\ \mathsf{c} = \mathsf{c} + 1$. A function definition:

$$\vdash\ \mathsf{f} \;=\; \lambda x_1\ \cdots\ x_n.\ t$$

can be written as:

$$\vdash\ \mathsf{f}\, x_1\ \cdots\ x_n \;=\; t$$

The HOL system currently permits the user to postulate arbitrary axioms when he builds a theory. This freedom is dangerous because inconsistent axioms can be introduced (e.g. by postulating $\vdash\ \mathsf{T} = \mathsf{F}$). As was shown by Russell and Whitehead [Hatcher], with suitable definitions, all of classical mathematics can be constructed from logic together with the assumption that there are infinitely many individuals (the Axiom of Infinity). It would thus appear reasonable to restrict the user to only making definitions and we eventually plan to do this.

6.2 Type definitions

Types denote sets. For example, the primitive type *bool* denotes the set of two truth-values and the primitive type *ind* denotes some infinite set of individuals. Compound types denote sets built by forming sets of functions. For example, $ind{\to}bool$ denotes the sets of functions from the set of individuals to the set of truth-values. Using the techniques described in Appendix B it is possible to represent any useful set as a subset of some set constructed from the truth-values and individuals. Unfortunately the representing sets are often quite complicated and it is useful to have some abstraction mechanism for hiding the details. The motivation for this is similar to the motivation for data abstraction in programming.

As an example, let us consider how we might represent times consisting of hours and minutes. Such a time can be represented by a pair $(hours, mins)$ where $hours$ and $mins$ are numbers. Now it turns out (see Appendix B) that numbers can be represented as a subset of the set individuals, and pairs $(x{:}\sigma_1, y{:}\sigma_2)$ can be represented as functions of type $\sigma_1{\to}\sigma_2{\to}bool$. Thus times can be represented as objects of type $ind{\to}ind{\to}bool$.

Suppose we want a function to increment the hour component of times. We might define a constant, $\mathsf{Inc\_Hour}$ say, of type $(ind{\to}ind{\to}bool){\to}(ind{\to}ind{\to}bool)$ to represent this. It would be nice if we could make this type more intelligible by somehow introducing a new type, $time$ say, so that $\mathsf{Inc\_Hour}$ had type $time{\to}time$. A simple approach would be to use abbreviations so that $time$ and $ind{\to}ind{\to}bool$ would be interchangeable. The problem with this is that there is no way of making explicit which uses of $ind{\to}ind{\to}bool$ represent times and which ones represent other things.

As another example consider places; these could also be represented as pairs of numbers (i.e. (m, n) specifies 2-D coordinates), so the type $ind{\to}ind{\to}bool$ could be the representing type for places also. One might thus introduce the abbreviation $place$ for $ind{\to}ind{\to}bool$. But then the function $\mathsf{Inc\_Hour}$ would be just as applicable to places as it is to times. Clearly one wants some way of indicating when something of type $ind{\to}ind{\to}bool$ is intended to be a place and when it is intended to be a time. This is achieved in HOL by keeping the types $ind{\to}ind{\to}bool$, $time$ and $place$ distinct and then introducing axioms that say that they are *isomorphic* (i.e. in one-to-one correspondence).

Types σ_1 and σ_2 are isomorphic if and only if there exist functions $f_1{:}\sigma_1{\to}\sigma_2$ and $f_2{:}\sigma_2{\to}\sigma_1$ (called *isomorphisms*) such that:

$$\vdash\ \forall x_1{:}\sigma_1.\ f_2(f_1\ x_1) = x_1$$
$$\vdash\ \forall x_2{:}\sigma_2.\ f_1(f_2\ x_2) = x_2$$

If σ_1 is a new type and σ_2 its representing type, then f_1 should be thought of as a *representation function* that maps elements of the new type to the corresponding elements of the old type that represent them. The function f_2 is the inverse to f_1 and can be thought of as an *abstraction function* mapping representations to the 'abstract' objects of the new type they represent.

To make the types $time$ and $place$ isomorphic to $ind{\to}ind{\to}bool$ we must introduce isomorphisms:

$$\mathsf{Rep\_Time} : time{\to}(ind{\to}ind{\to}bool)$$
$$\mathsf{Abs\_Time} : (ind{\to}ind{\to}bool){\to}time$$
$$\mathsf{Rep\_Place} : place{\to}(ind{\to}ind{\to}bool)$$
$$\mathsf{Abs\_Place} : (ind{\to}ind{\to}bool){\to}place$$

A Term of type $ind{\to}ind{\to}bool$ can be explicitly 'coerced' to a term of type $time$ or to a term of type $place$ by applying to it the appropriate abstraction function — i.e.

$\mathsf{Abs\_Time}$ or $\mathsf{Abs\_Place}$ respectively. If $\mathsf{Inc\_Hour}$ had type *time*→*time* then a term $\mathsf{Inc\_Hour}\ t$ would not be well-typed if t had type *ind*→*ind*→*bool*, but any term of the form $\mathsf{Inc\_Hour}(\mathsf{Abs\_Time}\ t)$ would be well-typed.

Usually one does not want to define a new type to be isomorphic to all of some existing type, but only to a subset of it. For example, one might only want pairs $(hours, mins)$ to be the representation of a time if $hours \leq 24$ and $mins \leq 60$. The subsets of the set of pairs of numbers corresponding to times and places can be specified by suitably defined predicates $\mathsf{Is\_Time}$ and $\mathsf{Is\_Place}$. If times are constrained as above but any pair of numbers can represent a place then:

$$\begin{array}{l} \vdash\ \mathsf{Is\_Time}(hours, mins)\ =\ (hours \leq 24)\ \wedge\ (mins \leq 60) \\ \vdash\ \mathsf{Is\_Place}(m, n)\ =\ \mathsf{T} \end{array}$$

Instead of requiring types *time* and *place* to be isomorphic to all of *ind*→*ind*→*bool* we really want them to be isomorphic to the subsets specified by the predicates $\mathsf{Is\_Time}$ and $\mathsf{Is\_Place}$ respectively. We show how to axiomatize this requirement shortly.

As well as defining types it is also convenient to be able to define type operators. For example, to represent pairs one would like to define a binary type operator *prod*. The way one does this is to use a representing type for $(\alpha_1, \alpha_2)prod$ that contains the type variables α_1 and α_2. As is explained in Appendix B, a suitable type for this purpose is $\alpha_1 \to \alpha_2 \to bool$.

Types in HOL must be non-empty to ensure that $\vdash\ (\forall x{:}\sigma.\ t) \supset (\exists x{:}\sigma.\ t)$ for every type σ. Thus one can only define a new type isomorphic to a subset specified by a predicate P, if $\vdash\ \exists x.\ P(x)$.

To summarize, a new type is defined by:

1. Specifying an existing type.

2. Specifying a subset of this type.

3. Proving that this subset is non-empty.

4. Specifying that the new type is isomorphic to this subset.

More formally, to define a new type $(\alpha_1, \ldots, \alpha_n)op$ one must:

1. Specify a type, σ_{op} say, called the *representing type*. This should only contain the type variables α_1, …, α_n. The type $(\alpha_1, \ldots, \alpha_n)op$ is intended to be isomorphic to a subset of σ_{op}.

2. Specify a term, P_{op} say, of type $\sigma_{op} \to bool$ called the *subset predicate*. This defines the subset of σ_{op} that $(\alpha_1, \ldots, \alpha_n)op$ is to be isomorphic to.

3. Prove $\vdash\ \exists x{:}\sigma_{op}.\ P_{op}\ x$.

4. Introduce a new constant, Rep_{op} say, of type $(\alpha_1, \ldots, \alpha_n)op \to \sigma_{op}$ called the *representation function*, together with appropriate axioms (see below), to specify the isomorphism from $(\alpha_1, \ldots, \alpha_n)op$ to the subset of σ_{op} determined by P_{op}. (We only need to take the representation function as primitive, the abstraction function can be defined as its inverse).

To specify Rep_{op} we must assert an axiom that says that it is a one-to-one mapping and also that it is onto the subset of σ_{op} determined by P_{op}.

To make this formal the theory `bool` (see below) provides a polymorphic constant $\mathsf{One\_One}$ defined by:

$$\vdash\ \mathsf{One\_One}\ =\ \lambda f{:}\alpha \to \beta.\ \forall x_1\ x_2.\ (f\ x_1 = f\ x_2) \Rightarrow (x_1 = x_2)$$

Thus $\mathsf{One\_One}\ f$ is true if and only if f is one-to-one. `bool` also provides a constant $\mathsf{Onto\_Subset}$ defined by:

$$\vdash\ \mathsf{Onto\_Subset}\ =\ \lambda f{:}\alpha \to \beta.\ \lambda P{:}\beta \to bool.\ \forall x{:}\beta.\ (P\ x) = (\exists x'{:}\alpha.\ x = f\ x')$$

Thus $\mathsf{Onto\_Subset}\ f\ P$ is true if and only if the range of f is the subset determined by P.

The axiom that characterizes Rep_{op} as an isomorphism from $(\alpha_1, \ldots, \alpha_n)op$ onto the subset of σ_{op} determined by P_{op} is:

$$\vdash\ (\mathsf{One\_One}\ \mathsf{Rep}_{op})\ \ \wedge\ \ (\mathsf{Onto\_Subset}\ \mathsf{Rep}_{op}\ P_{op})$$

Defining a new type $(\alpha_1, \ldots, \alpha_n)op$ in a theory `th` consists of introducing op as a new n-ary type operator of `th`, Rep_{op} as a new constant of `th` and the above axiom as a new axiom of `th`. Such a type definition is only valid if:

- op isn't already a type operator of `th`,

- Rep_{op} isn't already a constant of `th` and
- $\vdash\ \exists x{:}\sigma_{op}.\ P_{op}\ x$ is a theorem of `th`.

Examples of type definitions are given in Appendix B.

6.3 Inference rules

Inference rules are procedures for deriving new theorems. In the HOL system they are represented as functions in ML [Gordon *et al.* (78), Gordon (82)]. There are eight primitive inference rules, all other rules are derived from these and the axioms (see Appendix A for some example derivations). Below are listed the primitive inference rules in standard natural deduction notation. The metavariables t, t_1, t_2 etc. stand for arbitrary terms. The theorems above the horizontal line are called the *hypotheses* of the rule and the theorem below the line is called the *result*. Each rule says that its result can be deduced from its hypotheses, provided any restrictions mentioned below the rule hold. The first three rules below have no hypotheses, their results can always be deduced. The identifiers in square brackets are the names of the rules in the HOL system.

Assumption introduction [`ASSUME`]

$$\frac{}{t\ \vdash\ t}$$

Reflexivity [`REFL`]

$$\frac{}{\vdash\ t = t}$$

Beta-conversion [`BETA_CONV`]

$$\frac{}{\vdash\ (\lambda x.\ t_1)t_2 = t_1[t_2/x]}$$

- Where $t_1[t_2/x]$ is the resultof substituting t_2 for x in t_1, with the restriction that no free variables in t_2 become bound after substitution into t_1.

Substitution [SUBST]

$$\frac{\Gamma_1 \ \vdash\ t_1 = t_2 \qquad\qquad \Gamma_2 \ \vdash\ t[t_1]}{\Gamma_1 \cup \Gamma_2 \ \vdash\ t[t_2]}$$

- Where $t[t_1]$ denotes a term t with some free occurrences of t_1 singled out and $t[t_2]$ denotes the result of replacing these occurrences of t_1 by t_2, with the restriction that the context $t[\ ]$ must not bind any variable occurring free in either t_1 or t_2.

- $\Gamma_1 \cup \Gamma_2$ is the set union of Γ_1 and Γ_2.

Abstraction [ABS]

$$\frac{\Gamma \ \vdash\ t_1 = t_2}{\Gamma \ \vdash\ (\lambda x.\ t_1) = (\lambda x.\ t_2)}$$

- Provided x is not free in Γ.

Type instantiation [INST_TYPE]

$$\frac{\Gamma \ \vdash\ t}{\Gamma \ \vdash\ t[\sigma_1,\ \ldots\ ,\sigma_n/\alpha_1,\ \ldots\ ,\alpha_n]}$$

- Where $t[\sigma_1,\ \ldots\ ,\sigma_n/\alpha_1,\ \ldots\ ,\alpha_n]$ is the result of substituting in parallel the types $\sigma_1, \ldots, \sigma_n$ for type variables $\alpha_1, \ldots, \alpha_n$ in t, with the restriction that none of $\alpha_1, \ldots, \alpha_n$ occur in Γ.

Discharging an assumption [DISCH]

$$\frac{\Gamma \ \vdash\ t_2}{\Gamma - \{t_1\} \ \vdash\ t_1 \Rightarrow t_2}$$

- Where $\Gamma - \{t_1\}$ is the set subtraction of $\{t_1\}$ from Γ.

Modus Ponens [MP]

$$\frac{\Gamma_1 \ \vdash\ t_1 \Rightarrow t_2 \qquad\qquad \Gamma_2 \ \vdash\ t_1}{\Gamma_1 \cup \Gamma_2 \ \vdash\ t_2}$$

7 Semantics

In this section we give a very informal sketch of the intended semantics of the HOL logic.

The essential idea is that types denote sets and terms denote members of these sets. Only well-typed terms are considered meaningful. If term t has type σ then t should denote a member of the set denoted by σ.

The meaning of a type depends on the interpretation of the type variables (as sets) that it contains. A type σ containing type variables $\alpha_1, \ldots, \alpha_m$ denotes a function from m-tuples of sets to sets, such a function is not itself a set but is a class. For example, the type $\alpha{\rightarrow}\alpha$ denotes the 'class function' that maps a set X to the set of functions from X to X (i.e. $\alpha{\rightarrow}\alpha$ denotes $X \mapsto \{f \mid f : X{\rightarrow}X\}$).

Polymorphic constants are interpreted as functions of the interpretations of the type variables in their type. For example, the standard meaning of the constant $\mathsf{I}{:}\alpha{\rightarrow}\alpha$ is the function that maps a set X (the interpretation of α) to the identity function on X.

The meaning of a term depends on the interpretation of the constants, free variables and type variables in it. The interpretation of a term t with type variables $\alpha_1, \ldots, \alpha_m$ and free variables $x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n$ is a function from $m{+}n$-tuples of sets to sets. More specifically, it is a function from tuples $(X_1, \ldots, X_m, v_1, \ldots, v_n)$ where each X_i is a set and each v_i is a member of the interpretation of σ_i (where σ_i is interpreted with respect to the interpretation of $\alpha_1, \ldots, \alpha_m$ as $X_1, \ldots, X_m$). For example, the interpretation of $(\lambda x{:}\alpha.\ x)\ y$ with respect to the tuple (X, v) is v, where X is the interpretation of α and $v \in X$ is the interpretation of y (i.e. the term $(\lambda x{:}\alpha.\ x)\ y$ denotes $(X, v) \mapsto v$).

Type variables are regarded as implicitly universally quantified at the outermost level. Thus a theorem $\vdash\ (\lambda x{:}\alpha.\ x)\ y = y$ asserts that with respect to every interpretation of α as a (non-empty) set X the interpretation of $\lambda x{:}\alpha.\ x$ is a function which when applied to the interpretation, v say, of y yields v.

Type variables are really just ordinary variable of type 'set'. Polymorphic constants are just functions of such variables. It might be better to make this explicit in the syntax by, for example, forcing one to define I by:

$$\vdash\ \mathsf{I}(\alpha) = \lambda x{:}\alpha.\ x$$

rather than:

$$\vdash \mathsf{I} = \lambda x{:}\alpha.\ x$$

The syntax and semantics of type variables are currently being studied by several logicians. A closely related area is the theory of 'second order' λ-terms like $\lambda\alpha.\ \lambda x{:}\alpha.\ x$, perhaps such terms should be included in the HOL logic.

8 Theories

A *theory* consists of a set of types, type operators, constants, definitions, axioms and theorems. The usual definition of a theory in textbooks on mathematical logic is a bit different from the HOL notion of a theory. In particular, following LCF, the theorems in a HOL theory are just those that have been explicitly proved and saved by a user of the system. In logic, one usually says that a theory contains all the (possibly infinitely many) theorems that follow from the definitions; no terminological distinction is drawn between theorems that have actually been proved and those that could in principle be proved (i.e. that logically follow).

Theories can have other theories as *parents*; if `th1` is a parent of `th2` then all the types, constants, definitions, axioms and theorems of `th1` are available for use in `th2`. The structure with nodes consisting of theories and edges corresponding to parenthood relations is required to be a directed acyclic graph. If `th1` is a parent of `th2` we say `th2` is a *descendant* of `th1`. The theories that are built into the HOL system have the parenthood structure shown below (parents are drawn above their descendants).

```
                BOOL
                 |
                 |
       |---------|---------|
       |                   |
      IND                 PROD
       |                   |
       |                   |
      NUM                  |
       |                   |
       |                   |
   PRIM_REC                |
       |                   |
       |                   |
   ARITHMETIC              |
       |                   |
       |---------|---------|
                 |
                LIST
                 |
                 |
                HOL
```

The theories `bool` and `ind` are primitive and are described in detail below. All the other theories we need (see Appendix B) can be defined in terms of them.

8.1 The theory `bool`

The most basic theory is `bool`. This has a descendant theory `ind` that introduces the type *ind* of individuals together with the Axiom of Infinity that says there are infinitely many individuals. This axiom, together with the axioms in `bool` and the rules of inference of HOL, permits the development of all of classical mathematics.

The only primitive logical constants in HOL are $\Rightarrow$, $=$ and ε. The first two of these denote logical implication and equality, the third is Hilbert's ε-operator which is described below (Church uses an upside down ι instead of ε).

8.1.1 Hilbert's ε-operator

If $t[x]$ is a boolean term containing a free variable x of type σ, then the Hilbert-term $\varepsilon x.\ t[x]$ denotes some value of type σ, a say, such that $t[a]$ is true. For example, the term $\varepsilon n.\ n < 10$ denotes some unspecified number less than 10 and the term $\varepsilon n.\ (n^2 = 25) \wedge (n \geq 0)$ denotes 5.

If there is no a of type σ such that $t[a]$ is true then $\varepsilon x.\ t[x]$ denotes a fixed but unspecified value of type σ. For example, $\varepsilon n.\ \neg(n = n)$ denotes an unspecified number. One of the axioms of HOL states that if $\exists x.\ t[x]$ is true then it follows that $t[\varepsilon x.\ t[x]]$ is true also.

It must be admitted that the ε-operator looks rather suspicious. For a thorough discussion of it see [Leisenring]. It is useful for naming things one knows to exist but have no name. For example, the Peano-Lawvere axiom asserts that given a number n_0 and a function $f{:}\mathit{num}{\to}\mathit{num}$, there exists a unique sequence s defined recursively by:

$$(s(0) = n_0)\ \wedge\ (\forall n.\ s(n+1) = f(s(n)))$$

Using the ε-operator we can define a function, Rec say, that returns s when given the pair (n_0, f) as an argument:

$$\mathsf{Rec}(n_0, f) \quad = \quad \varepsilon s.\ (s(0) = n_0) \wedge (\forall n.\ s(n+1) = f(s(n)))$$

$\mathsf{Rec}(n_0, f)$ denotes the unique sequence whose existence is asserted by the Peano-Lawvere Axiom. It follows from this axiom that:

$$(\mathsf{Rec}(n_0, f)0\ =\ n_0)\ \wedge\ (\forall n.\ \mathsf{Rec}(n_0, f)(n+1) = f(\mathsf{Rec}(n_0, f)n))$$

Many things that are normally primitive can be defined using the ε-operator. For example, the conditional term $\mathsf{Cond}\ t\ t_1\ t_2$ (meaning "if t then t_1 else t_2") can be defined by:

$$\mathsf{Cond}\ t\ t_1\ t_2 \quad = \quad \varepsilon x.\ ((t = \mathsf{T}) \Rightarrow (x = t_1))\ \wedge\ ((t = \mathsf{F}) \Rightarrow (x = t_2))$$

One can use the ε-operator to simulate λ-abstraction: if the variable f does not occur in the term t, then the function $\lambda x.\ t$ is equivalent to $\varepsilon f.\ \forall x.\ f(x) = t$ ("the function f such that $f(x) = t$ for all x"). This idea can be used to create functional abstractions that cannot be expressed with simple λ-terms. For example, the factorial function is denoted by:

$$\varepsilon f.\ \forall n.\ (f(0) = 1)\quad \wedge \quad (f(n+1) = (n+1) \times f(n))$$

Terms like this can be used to simulate the kind of pattern matching mechanisms found in programming languages like Hope [Burstall *et al.*] and Standard ML [Milner (84)].

The inclusion of ε-terms into HOL 'builds in' the Axiom of Choice [Hatcher]. In Set Theory, the Axiom of Choice states that if $\mathcal{S}$ is a family of sets then there exists a function, Choose say, such that for each non-empty $X \in \mathcal{S}$ we have $\mathsf{Choose}(X) \in X$.

As sets are not primitive in HOL, we must reformulate Choose to act on the characteristic functions of sets rather than sets themselves. The characteristic function of a set X is the function f_X with range $\{\mathsf{T}, \mathsf{F}\}$ defined by $f_X(x) = \mathsf{T}$ if and only if $x \in X$. If P is any function with range $\{\mathsf{T}, \mathsf{F}\}$, we call P *non-empty* if for some x it is the case that $P(x) = \mathsf{T}$ (so f_X is non-empty if and only if X is non-empty). The HOL version of the Axiom of Choice asserts that there exists a function, Select say, such that if P is a non-empty function with range $\{\mathsf{T}, \mathsf{F}\}$ then $P(\mathsf{Select}(P)) = \mathsf{T}$. Intuitively $\mathsf{Select}\ P$ is just $\mathsf{Choose}\{x \mid P\ x = \mathsf{T}\}$.

Hilbert's ε-operator is a binder that denotes Select. More precisely ε is a binder with type $(\alpha{\to}bool){\to}\alpha$ which is interpreted so that if P has type $\sigma{\to}bool$ then:

- $\$\varepsilon(P)$ denotes some fixed (but unknown) value x such that $P(x) = \mathsf{T}$ if such a value exists;
- if no such value exists (i.e. $P(x) = \mathsf{F}$ for all x) then $\$\varepsilon(P)$ denotes some unspecified value in the set denoted by σ.

Having ε-terms forces every type to be non-empty because the term $\varepsilon x{:}\sigma.\mathsf{T}$ always denotes a member of σ.

8.1.2 Definitions of the logical constants

There are only three primitive logical constants in HOL, namely $\Rightarrow$, $=$ and ε. These are all constants of the theory `bool`. Within this theory the other logical constants can be defined by:

$$
\begin{array}{lcl}
\vdash \mathsf{T} & = & ((\lambda x.\ x) = (\lambda x.\ x)) \\
\vdash \$\forall & = & \lambda P.\ \ P = (\lambda x.\ \mathsf{T}) \\
\vdash \$\exists & = & \lambda P.\ \ P(\$\varepsilon\ P) \\
\vdash \mathsf{F} & = & \forall b.\ \ b \\
\vdash \neg & = & \lambda b.\ \ b \Rightarrow \mathsf{F} \\
\vdash \$\wedge & = & \lambda b_1\ b_2.\ \forall b.\ (b_1 \Rightarrow (b_2 \Rightarrow b)) \Rightarrow b \\
\vdash \$\vee & = & \lambda b_1\ b_2.\ \forall b.\ (b_1 \Rightarrow b) \Rightarrow ((b_2 \Rightarrow b) \Rightarrow b) \\
\vdash \$\equiv & = & \lambda b_1\ b_2.\ (b_1 \Rightarrow b_2) \wedge (b_2 \Rightarrow b_1) \\
\vdash \$\exists! & = & \lambda P.\ (\$\exists\ P) \wedge (\forall x\ y.\ (P\ x) \wedge (P\ y) \Rightarrow (x = y))
\end{array}
$$

These definitions may seem rather obscure, but it turns out that all the usual properties can be derived from them starting from the rules of inference and the following axioms:

$$\begin{array}{l}
\vdash \forall b.\ (b = \mathsf{T}) \vee (b = \mathsf{F}) \\
\vdash \forall b_1\ b_2.\ (b_1 \Rightarrow b_2) \Rightarrow (b_2 \Rightarrow b_1) \Rightarrow (b_1 = b_2) \\
\vdash \forall f.\ (\lambda x.\ f\ x) = f \\
\vdash \forall P\ x.\ P\ x \Rightarrow P(\$\varepsilon\ P)
\end{array}$$

These are the only non-definitional axioms in the theory `bool`. In Appendix A we show how the standard rules of logic can be derived from these axioms, the definitions of the logical constants and the primitive rules of inference. The only other non-definitional axiom in HOL is the Axiom of Infinity which is part of the theory `ind`.

8.1.3 Other constants in the theory `bool`

It is convenient to include in `bool` the definitions of One_One and Onto_Subset that are used when making type definitions, as well as some other well-known and useful constants.

$$\begin{array}{l}
\vdash \mathsf{One_One}\ =\ \lambda f{:}\alpha{\to}\beta.\ \forall x_1\ x_2.\ (f\ x_1 = f\ x_2) \Rightarrow (x_1 = x_2) \\
\vdash \mathsf{Onto_Subset}\ =\ \lambda f{:}\alpha{\to}\beta.\ \lambda P.\ \forall x.\ (P\ x) = (\exists x'{:}\alpha.\ x = f\ x') \\
\vdash \mathsf{Onto}\ =\ \lambda f{:}\alpha{\to}\beta.\ \forall y.\ \exists x.\ y = f\ x \\
\vdash \mathsf{Inv}\ =\ \lambda f{:}\alpha{\to}\beta.\ \lambda y.\ \varepsilon x.\ y = f\ x \\
\vdash \$\mathsf{o}\ =\ \lambda f{:}\beta{\to}\gamma.\ \lambda g{:}\alpha{\to}\beta.\ \lambda x.\ f(g\ x) \qquad\qquad (\mathsf{o}\text{ is an infix}) \\
\vdash \mathsf{I}\ =\ \lambda x{:}\alpha.\ x
\end{array}$$

The following theorems follow from these definitions:

$$\begin{array}{lll}
\mathsf{One_One}\ f & \vdash & (\mathsf{Inv}\ f)\ \mathsf{o}\ f\ =\ \mathsf{I} \\
\mathsf{Onto}\ f & \vdash & f\ \mathsf{o}\ (\mathsf{Inv}\ f)\ =\ \mathsf{I}
\end{array}$$

The definition of the conditional function also included in `bool`:

$$\vdash \mathsf{Cond}\ t\ t_1\ t_2\ =\ \varepsilon x.\ ((t = \mathsf{T}) \Rightarrow (x = t_1))\ \wedge\ ((t = \mathsf{F}) \Rightarrow (x = t_2))$$

From this definition it is straightforward to deduce that:

$$\vdash \forall x_1\ x_2.\ (\mathsf{Cond}\ \mathsf{T}\ x_1\ x_2 = x_1)\ \wedge\ (\mathsf{Cond}\ \mathsf{F}\ x_1\ x_2 = x_2)$$

There is a special syntax for conditionals: $(t \to t_1 \mid t_2)$ means $\mathsf{Cond}\ t\ t_1\ t_2$.

8.2 The theory `ind`

If there are m distinct elements of type σ_1 and n distinct elements of type σ_2 then there are n^m of type $\sigma_1 \rightarrow \sigma_2$. Thus, starting with the two element type *bool* one can only generate types containing finitely many elements using $\rightarrow$. There are infinitely many numbers, so there is no hope of constructing a representing type for numbers from *bool* and $\rightarrow$. To get over this problem we postulate a new primitive type *ind* that has infinitely many distinct elements. The theory `ind` introduces this type *ind* and has one axiom, called the Axiom of Infinity:

$$\vdash \ \exists f{:}ind{\rightarrow}ind.\ \mathsf{One\_One}\ f\ \wedge\ \neg(\mathsf{Onto}\ f)$$

It may not be obvious that this implies there are infinitely many distinct elements of type *ind*, to see that it does first define:

$$\vdash \ \mathsf{Suc\_Rep}\ =\ \varepsilon f{:}ind{\rightarrow}ind.\ \mathsf{One\_One}\ f\ \wedge\ \neg(\mathsf{Onto}\ f)$$

Then it follows from the Axiom of Infinity that:

$$\vdash \ \mathsf{One\_One}\ \mathsf{Suc\_Rep}\ \wedge\ \neg(\mathsf{Onto}\ \mathsf{Suc\_Rep})$$

From the second conjunct of this and the definition of Onto it follows that if we define:

$$\vdash \ \mathsf{Zero\_Rep}\ =\ \varepsilon y{:}ind.\ \forall x.\ \neg(y = \mathsf{Suc\_Rep}\ x)$$

then:

$$\vdash \ \forall x.\ \neg(\mathsf{Zero\_Rep} = \mathsf{Suc\_Rep}\ x)$$

From this and the definition of One_One it is easy to show that the sequence of terms Zero_Rep, Suc_Rep Zero_Rep, Suc_Rep(Suc_Rep Zero_Rep) etc. are all distinct. Thus, if we denote the result of applying Suc_Rep to Zero_Rep n times by $\mathsf{Suc\_Rep}^n$ Zero_Rep, then the set of elements of this form with $n = 1, 2, 3\ldots$ is an infinite set. Thus the Axiom of Infinity does indeed imply that there are infinitely many distinct elements of type *ind*.

9 Acknowledgements

The use of higher-order logic for hardware specification and verification has been pioneered by Keith Hanna [Hanna-Daeche]. The HOL system is quite similar to his VERITAS system.

I was inspired to move from an *ad hoc* special purpose logic (namely LCF_LSM [Gordon (83)]) to 'pure logic' by the elegant work of Ben Moszkowski on using Interval Temporal Logic (ITL) for hardware description [Halpern]. One of the design goals of the HOL logic was to construct a framework to support reasoning in ITL. How this is done will be the subject of a future paper.

In formulating details of the HOL logic I was helped by advice from the logicians Mike Fourman and Martin Hyland. In particular, Mike Fourman explained to me how numbers could be represented (see Appendix B) and how types should be defined.

The HOL system is based on Cambridge LCF [Paulson] which, in turn, evolved from Robin Milner's Edinburgh LCF [Gordon79]. Many ideas (and much code) from LCF are incorporated in HOL.

I have had many valuable discussions with the various users of HOL. These include Albert Camilleri, Nives Chaplin, Inder Dhingra, John Herbert, Tom Melham and Edmund Ronald from Cambridge, and Jeff Joyce from the University of Calgary. Several of these people provided criticisms and comments on a draft of this paper.

Francisco Corella (of Schlumberger Palo Alto Research) pointed out several errors in the first version of this paper as well as making some good suggestions for improvements.

10 References

[Andrews]
P. Andrews, D. Miller, E. Longini Cohen and F. Pfenning. *Automating Higher Order Logic*. Contemporary Mathematics 29, 1984.

[Boyer-Moore]
R. Boyer and J Moore. *A Computational Logic*. Academic Press. New York, 1979.

[Burstall]

R. Burstall, D. MacQueen and D. Sannella. *HOPE: An Experimental Applicative Language*. Report CSR-62-80, Computer Science Department, Edinburgh University, 1980.

[Church]

A. Church. *A Formulation of the Simple Theory of Types*. Journal of Symbolic Logic 5, 1940.

[Gordon78]

M. Gordon, R. Milner, L. Morris, M. Newey and C. Wadsworth. *A metalanguage for interactive proof in LCF*. In the Proceedings of the Fifth ACM SIGACT-SIGPLAN Conference on *Principles of Programming Languages*. Tucson, Arizona, 1978.

[Gordon79]

M. Gordon, R. Milner and C. Wadsworth. *Edinburgh LCF: A mechanised logic of computation*. Lecture Notes in Computer Science No. 78, Springer-Verlag, 1979.

[Gordon (82)]

M. Gordon. *Representing a Logic in the LCF Metalanguage*. In *Tools and Notions for Program Construction* (ed. D. Neel), Cambridge University Press, 1982.

[Gordon (83)]

M. Gordon. *LCF_LSM*. University of Cambridge Computer Laboratory Technical Report No. 41, 1983.

[Halpern]

J. Halpern, Z. Manna and B. Moszkowski. *A hardware semantics based on temporal intervals*. In the proceedings of the *10-th International Colloquium on Automata, Languages and Programming*, Barcelona, Spain, 1983.

[Hanna-Daeche]

F. K. Hanna and N. Daeche. *Specification and Verification using Higher-Order Logic*. Proceedings of the *7th International Conference on Computer Hardware Design Languages*. Tokyo, 1985.

[Hatcher]

W. Hatcher. *The Logical Foundations of Mathematics*. Pergamon Press, 1982.

[Ketonen-Weening]

J. Ketonen and J. Weening. *EKL - An Interactive Proof Checker.* Stanford University, 1983.

[Leisenring]

A. Leisenring. *Mathematical Logic and Hilbert's ε-Symbol.* Macdonald & Co. Ltd. London, 1969.

[MacLane and Birkhoff]

S. MacLane and G Birkhoff. *Algebra.* The Macmillan Company, 1967.

[Milner (78)]

R. Milner. *A Theory of Type Polymorphism in Programming.* Journal of Computer and System Sciences 17, 1978.

[Milner (84)]

R. Milner. *A Proposal for Standard ML.* Proceedings of the *1984 ACM Symposium on LISP and Functional Programming*, Austin, Texas, 1984.

[Paulson]

L. Paulson. *The Revised Logic PPLAMBDA: A Reference Manual.* University of Cambridge Computer Laboratory Technical Report Number 36, 1983. *A Higher Order Implementation of Rewriting.* Science of Computer Programming, 1983. *Tactics and Tacticals in Cambridge LCF.* University of Cambridge Computer Laboratory Technical Report Number 39, 1983.

A Derived Rules and Theorems

We outline below how the standard rules of logic can be derived from the axioms and definitions in `bool` using the primitive inference rules of the HOL logic.

The derivations that follow consist of sequences of numbered steps each of which:

- is an axiom, or
- a hypothesis of the rule being derived, or
- follows from preceding steps by a rule of inference (either primitive or previously derived).

Theorems will be treated as rules that have no hypotheses (thus a derivation of a theorem is like the derivation of a rule but without any hypotheses). Note that there are rules without hypotheses that are more general than theorems. For example, for any terms t_1 and t_2 the theorem $\vdash\ (\lambda x.\ t_1)t_2 = t_1[t_2/x]$ follows from `BETA_CONV`. This rule thus generates a theorem for each pair of terms t_1, t_2 and is thus equivalent to infinitely many theorems. There is no single theorem in the HOL logic equivalent to `BETA_CONV`.

A.1 Adding an assumption [`ADD_ASSUM`]

$$\frac{\Gamma\ \vdash\ t}{\Gamma,\ t'\ \vdash\ t}$$

1. $t'\ \vdash\ t'$ [`ASSUME`]
2. $\Gamma\ \vdash\ t$ [Hypothesis]
3. $\Gamma\ \vdash\ t' \Rightarrow t$ [`DISCH` 2]
4. $\Gamma,\ t'\ \vdash\ t$ [`MP` 3,1]

A.2 Undischarging [`UNDISCH`]

$$\frac{\Gamma\ \vdash\ t_1 \Rightarrow t_2}{\Gamma,\ t_1\ \vdash\ t_2}$$

1. $t_1 \vdash t_1$ [ASSUME]

2. $\Gamma \vdash t_1 \Rightarrow t_2$ [Hypothesis]

3. $\Gamma,\ t_1 \vdash t_2$ [MP 2,1]

A.3 Symmetry of equality [SYM]

$$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash t_2 = t_1}$$

1. $\Gamma \vdash t_1 = t_2$ [Hypothesis]

2. $\vdash t_1 = t_1$ [REFL]

3. $\Gamma \vdash t_2 = t_1$ [SUBST 1,2]

A.4 Transitivity of equality [TRANS]

$$\frac{\Gamma_1 \vdash t_1 = t_2 \qquad\qquad \Gamma_2 \vdash t_2 = t_3}{\Gamma_1 \cup \Gamma_2 \vdash t_1 = t_3}$$

1. $\Gamma_2 \vdash t_2 = t_3$ [Hypothesis]

2. $\Gamma_1 \vdash t_1 = t_2$ [Hypothesis]

3. $\Gamma_1 \cup \Gamma_2 \vdash t_1 = t_3$ [SUBST 1,2]

A.5 Application of a term to a theorem [AP_TERM]

$$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash t\ t_1 = t\ t_2}$$

1. $\Gamma \vdash t_1 = t_2$ [Hypothesis]

2. $\vdash t\ t_1 = t\ t_1$ [REFL]

3. $\Gamma \vdash t\ t_1 = t\ t_2$ [SUBST 1,2]

A.6 Application of a theorem to a term [AP_THM]

$$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash t_1\ t = t_2\ t}$$

1. $\Gamma \vdash t_1 = t_2$ [Hypothesis]
2. $\vdash t_1\ t = t_1\ t$ [REFL]
3. $\Gamma \vdash t_1\ t = t_2\ t$ [SUBST 1,2]

A.7 Modus Ponens for equality [EQ_MP]

$$\frac{\Gamma_1 \vdash t_1 = t_2 \qquad\qquad \Gamma_2 \vdash t_1}{\Gamma_1 \cup \Gamma_2 \vdash t_2}$$

1. $\Gamma_1 \vdash t_1 = t_2$ [Hypothesis]
2. $\Gamma_2 \vdash t_1$ [Hypothesis]
3. $\Gamma_1 \cup \Gamma_2 \vdash t_2$ [SUBST 1,2]

A.8 Implication from equality [EQ_IMP_RULE]

$$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash t_1 \Rightarrow t_2 \qquad\qquad \Gamma \vdash t_2 \Rightarrow t_1}$$

1. $\Gamma \vdash t_1 = t_2$ [Hypothesis]
2. $t_1 \vdash t_1$ [ASSUME]
3. $\Gamma, t_1 \vdash t_2$ [EQ_MP 1,2]
4. $\Gamma \vdash t_1 \Rightarrow t_2$ [DISCH 3]
5. $\Gamma \vdash t_2 = t_1$ [SYM 1]
6. $t_2 \vdash t_2$ [ASSUME]
7. $\Gamma, t_2 \vdash t_1$ [EQ_MP 5,6]
8. $\Gamma \vdash t_2 \Rightarrow t_1$ [DISCH 7]
9. $\Gamma \vdash t_1 \Rightarrow t_2$ and $\Gamma \vdash t_2 \Rightarrow t_1$ [4,8]

A.9 T-Introduction [TRUTH]

$$\vdash\ \mathsf{T}$$

1. $\vdash\ \mathsf{T} = ((\lambda x.\ x) = (\lambda x.\ x))$ [Definition of T]
2. $\vdash\ ((\lambda x.\ x) = (\lambda x.\ x)) = \mathsf{T}$ [SYM 1]
3. $\vdash\ (\lambda x.\ x) = (\lambda x.\ x)$ [REFL]
4. $\vdash\ \mathsf{T}$ [EQ_MP 2,3]

A.10 Equality-with-T elimination [EQT_ELIM]

$$\frac{\Gamma\ \vdash\ t = \mathsf{T}}{\Gamma\ \vdash\ t}$$

1. $\Gamma\ \vdash\ t = \mathsf{T}$ [Hypothesis]
2. $\Gamma\ \vdash\ \mathsf{T} = t$ [SYM 1]
3. $\vdash\ \mathsf{T}$ [TRUTH]
4. $\Gamma\ \vdash\ t$ [EQ_MP 2,3]

A.11 Specialization ($\forall$-elimination) [SPEC]

$$\frac{\Gamma\ \vdash\ \forall x.\ t}{\Gamma\ \vdash\ t[t'/x]}$$

- $t[t'/x]$ denotes the result of substituting t' for free occurrences of x in t, with the restriction that no free variables in t' become bound after substitution.

1. $\vdash\ \forall = (\lambda P.\ P = (\lambda x.\ \mathsf{T}))$ [INST_TYPE applied to the definition of $\forall$]
2. $\Gamma\ \vdash\ \forall(\lambda x.\ t)$ [Hypothesis]
3. $\Gamma\ \vdash\ (\lambda P.\ P = (\lambda x.\ \mathsf{T}))(\lambda x.\ t)$ [SUBST 1,2]
4. $\vdash\ (\lambda P.\ P = (\lambda x.\ \mathsf{T}))(\lambda x.\ t) = ((\lambda x.\ t) = (\lambda x.\ \mathsf{T}))$ [BETA_CONV]
5. $\Gamma\ \vdash\ (\lambda x.\ t) = (\lambda x.\ \mathsf{T})$ [EQ_MP 4,3]

6. $\Gamma \vdash (\lambda x.\ t)\ t' = (\lambda x.\ \mathsf{T})\ t'$ [AP_THM 5]

7. $\vdash (\lambda x.\ t)\ t' = t[t'/x]$ [BETA_CONV]

8. $\Gamma \vdash t[t'/x] = (\lambda x.\ t)\ t'$ [SYM 7]

9. $\Gamma \vdash t[t'/x] = (\lambda x.\ \mathsf{T})\ t'$ [TRANS 8,6]

10. $\vdash (\lambda x.\ \mathsf{T})\ t' = \mathsf{T}$ [BETA_CONV]

11. $\Gamma \vdash t[t'/x] = \mathsf{T}$ [TRANS 9,10]

12. $\Gamma \vdash t[t'/x]$ [EQT_ELIM 11]

A.12 Equality-with-T introduction [EQT_INTRO]

$$\frac{\Gamma \vdash t}{\Gamma \vdash t = \mathsf{T}}$$

1. $\vdash \forall b_1\ b_2.\ (b_1 \Rightarrow b_2) \Rightarrow (b_2 \Rightarrow b_1) \Rightarrow (b_1 = b_2)$ [Axiom]

2. $\vdash \forall b_2.\ (t \Rightarrow b_2) \Rightarrow (b_2 \Rightarrow t) \Rightarrow (t = b_2)$ [SPEC 1]

3. $\vdash (t \Rightarrow \mathsf{T}) \Rightarrow (\mathsf{T} \Rightarrow t) \Rightarrow (t = \mathsf{T})$ [SPEC 2]

4. $\vdash \mathsf{T}$ [TRUTH]

5. $\vdash t \Rightarrow \mathsf{T}$ [DISCH 4]

6. $\vdash (\mathsf{T} \Rightarrow t) \Rightarrow (t = \mathsf{T})$ [MP 3,5]

7. $\Gamma \vdash t$ [Hypothesis]

8. $\Gamma \vdash \mathsf{T} \Rightarrow t$ [DISCH 7]

9. $\Gamma \vdash t = \mathsf{T}$ [MP 6,8]

A.13 Generalization ($\forall$-introduction) [GEN]

$$\frac{\Gamma \vdash t}{\Gamma \vdash \forall x.\ t}$$

- Where x is not free in Γ.

1. $\Gamma \vdash t$ [Hypothesis]

2. $\Gamma \vdash t = \mathsf{T}$ [`EQT_INTRO` 1]

3. $\Gamma \vdash (\lambda x.\ t) = (\lambda x.\ \mathsf{T})$ [`ABS` 2]

4. $\vdash \forall(\lambda x.\ t) = \forall(\lambda x.\ t)$ [`REFL`]

5. $\vdash \forall = (\lambda P.\ P = (\lambda x.\ \mathsf{T}))$ [`INST_TYPE` applied to the definition of $\forall$]

6. $\vdash \forall(\lambda x.\ t) = (\lambda P.\ P = (\lambda x.\ \mathsf{T}))(\lambda x.\ t)$ [`SUBST` 5,4]

7. $\vdash (\lambda P.\ P = (\lambda x.\ \mathsf{T}))(\lambda x.\ t) = ((\lambda x.\ t) = (\lambda x.\ \mathsf{T}))$ [`BETA_CONV`]

8. $\vdash \forall(\lambda x.\ t) = ((\lambda x.\ t) = (\lambda x.\ \mathsf{T}))$ [`TRANS` 6,7]

9. $\vdash ((\lambda x.\ t) = (\lambda x.\ \mathsf{T})) = \forall(\lambda x.\ \mathsf{T})$ [`SYM` 8]

10. $\Gamma \vdash \forall(\lambda x.\ t)$ [`EQ_MP` 9,3]

A.14 Simple α-conversion [`SIMPLE_ALPHA`]

$$\vdash (\lambda x_1.\ t\ x_1) = (\lambda x_2.\ t\ x_2)$$

- Where neither x_1 nor x_2 occur free in t.

1. $\vdash (\lambda x_1.\ t\ x_1)\ x = t\ x$ [`BETA_CONV`]

2. $\vdash (\lambda x_2.\ t\ x_2)\ x = t\ x$ [`BETA_CONV`]

3. $\vdash t\ x = (\lambda x_2.\ t\ x_2)\ x$ [`SYM` 2]

4. $\vdash (\lambda x_1.\ t\ x_1)\ x = (\lambda x_2.\ t\ x_2)\ x$ [`TRANS` 1,3]

5. $\vdash (\lambda x.\ (\lambda x_1.\ t\ x_1)\ x) = (\lambda x.\ (\lambda x_2.\ t\ x_2)\ x)$ [`ABS` 4]

6. $\vdash \forall f.\ (\lambda x.\ f\ x) = f$ [Appropriately type-instantiated axiom]

7. $\vdash (\lambda x.\ (\lambda x_1.\ t\ x_1) x) = \lambda x_1.\ t\ x_1$ [`SPEC` 6]

8. $\vdash (\lambda x.\ (\lambda x_2.\ t\ x_2) x) = \lambda x_2.\ t\ x_2$ [`SPEC` 6]

9. $\vdash (\lambda x_1.\ t\ x_1) = (\lambda x.\ (\lambda x_1.\ t\ x_1) x)$ [`SYM` 7]

10. $\vdash (\lambda x_1.\ t\ x_1) = (\lambda x.\ (\lambda x_2.\ t\ x_2) x)$ [`TRANS` 9,5]

11. $\vdash (\lambda x_1.\ t\ x_1) = (\lambda x_2.\ t\ x_2)$ [`TRANS` 10,8]

A.15 η-conversion [ETA_CONV]

$$\vdash\ (\lambda x'.\ t\ x') = t$$

- Where x' does not occur free in t (we use x' rather than just x to motivate the use of SIMPLE_ALPHA in the derivation below).

1. $\vdash\ \forall f.\ (\lambda x.\ f\ x) = f$ [Appropriately type-instantiated axiom]
2. $\vdash\ (\lambda x.\ t\ x) = t$ [SPEC 1]
3. $\vdash\ (\lambda x'.\ t\ x') = (\lambda x.\ t\ x)$ [SIMPLE_ALPHA]
4. $\vdash\ (\lambda x'.\ t\ x') = t$ [TRANS 3,2]

A.16 Extensionality [EXT]

$$\frac{\Gamma\ \vdash\ \forall x.\ t_1\ x = t_2\ x}{\Gamma\ \vdash\ t_1 = t_2}$$

- Where x is not free in Γ, t_1 or t_2.

1. $\Gamma\ \vdash\ \forall x.\ t_1\ x = t_2\ x$ [Hypothesis]
2. $\Gamma\ \vdash\ t_1\ x = t_2\ x$ [SPEC 1]
3. $\Gamma\ \vdash\ (\lambda x.\ t_1\ x) = (\lambda x.\ t_2\ x)$ [ABS 2]
4. $\vdash\ (\lambda x.\ t_1\ x) = t_1$ [ETA_CONV]
5. $\vdash\ t_1 = (\lambda x.\ t_1\ x)$ [SYM 4]
6. $\Gamma\ \vdash\ t_1 = (\lambda x.\ t_2\ x)$ [TRANS 5,3]
7. $\vdash\ (\lambda x.\ t_2\ x) = t_2$ [ETA_CONV]
8. $\Gamma\ \vdash\ t_1 = t_2$ [TRANS 6,7]

A.17 ε-introduction [`SELECT_INTRO`]

$$\frac{\Gamma \vdash t_1\ t_2}{\Gamma \vdash t_1(\varepsilon\ t_1)}$$

1. $\vdash \forall P\ x.\ P\ x \Rightarrow P(\varepsilon\ P)$ [Suitably type-instantiated axiom]
2. $\vdash t_1\ t_2 \Rightarrow t_1(\varepsilon\ t_1)$ [`SPEC` 1 (twice)]
3. $\Gamma \vdash t_1\ t_2$ [Hypothesis]
4. $\Gamma \vdash t_1(\varepsilon\ t_1)$ [`MP` 2,3]

A.18 ε-elimination [`SELECT_ELIM`]

$$\frac{\Gamma_1 \vdash t_1(\varepsilon\ t_1) \qquad \Gamma_2,\ t_1\ v \vdash t}{\Gamma_1 \cup \Gamma_2 \vdash t}$$

- Where v occurs nowhere except in the assumption $t_1\ v$ of the second hypothesis.

1. $\Gamma_2,\ t_1\ v \vdash t$ [Hypothesis]
2. $\Gamma_2 \vdash t_1\ v \Rightarrow t$ [`DISCH` 1]
3. $\Gamma_2 \vdash \forall v.\ t_1\ v \Rightarrow t$ [`GEN` 2]
4. $\Gamma_2 \vdash t_1(\varepsilon\ t_1) \Rightarrow t$ [`SPEC` 3]
5. $\Gamma_1 \vdash t_1(\varepsilon\ t_1)$ [Hypothesis]
6. $\Gamma_1 \cup \Gamma_2 \vdash t$ [`MP` 4,5]

A.19 $\exists$-introduction [`EXISTS`]

$$\frac{\Gamma \vdash t_1[t_2]}{\Gamma \vdash \exists x.\ t_1[x]}$$

- Where $t_1[t_2]$ denotes a term t_1 with some free occurrences of t_2 singled out, and $t_1[x]$ denotes the result of replacing these occurrences of t_1 by x, subject to the restriction that x doesn't become bound after substitution.

1. $\vdash (\lambda x.\ t_1[x])t_2 = t_1[t_2]$ [`BETA_CONV`]

2. $\vdash\ t_1[t_2] = (\lambda x.\ t_1[x])t_2$ [`SYM` 1]

3. $\Gamma\ \vdash\ t_1[t_2]$ [Hypothesis]

4. $\Gamma\ \vdash\ (\lambda x.\ t_1[x])t_2$ [`EQ_MP` 2,3]

5. $\Gamma\ \vdash\ (\lambda x.\ t_1[x])(\varepsilon(\lambda x.\ t_1[x]))$ [`SELECT_INTRO` 4]

6. $\vdash\ \exists = \lambda P.\ P(\varepsilon\ P)$ [`INST_TYPE` applied to the definition of $\exists$]

7. $\vdash\ \exists(\lambda x.\ t_1[x]) = (\lambda P.\ P(\varepsilon\ P))(\lambda x.\ t_1[x])$ [`AP_THM` 6]

8. $\vdash\ (\lambda P.\ P(\varepsilon\ P))(\lambda x.\ t_1[x]) = (\lambda x.\ t_1[x])(\varepsilon(\lambda x.\ t_1[x]))$ [`BETA_CONV`]

9. $\vdash\ \exists(\lambda x.\ t_1[x]) = (\lambda x.\ t_1[x])(\varepsilon(\lambda x.\ t_1[x]))$ [`TRANS` 7,8]

10. $\vdash\ (\lambda x.\ t_1[x])(\varepsilon(\lambda x.\ t_1[x])) = \exists(\lambda x.\ t_1[x])$ [`SYM` 9]

11. $\Gamma\ \vdash\ \exists(\lambda x.\ t_1[x])$ [`EQ_MP` 10,5]

A.20 $\exists$-elimination [`CHOOSE`]

$$\frac{\Gamma_1\ \vdash\ \exists x.\ t[x] \qquad\qquad \Gamma_2,\ t[v]\ \vdash\ t'}{\Gamma_1 \cup \Gamma_2\ \vdash\ t'}$$

- Where $t[v]$ denotes a term t with some free occurrences of the variable v singled out, and $t[x]$ denotes the result of replacing these occurrences of v by x, subject to the restriction that x doesn't become bound after substitution.

1. $\vdash\ \exists = \lambda P.\ P(\varepsilon\ P)$ [`INST_TYPE` applied to the definition of $\exists$]

2. $\vdash\ \exists(\lambda x.\ t[x]) = (\lambda P.\ P(\varepsilon\ P))(\lambda x.\ t[x])$ [`AP_THM` 1]

3. $\Gamma_1\ \vdash\ \exists(\lambda x.\ t[x])$ [Hypothesis]

4. $\Gamma_1\ \vdash\ (\lambda P.\ P(\varepsilon\ P))(\lambda x.\ t[x])$ [`EQ_MP` 2,3]

5. $\vdash\ (\lambda P.\ P(\varepsilon\ P))(\lambda x.\ t[x]) = (\lambda x.\ t[x])(\varepsilon(\lambda x.\ t[x]))$ [`BETA_CONV`]

6. $\Gamma_1\ \vdash\ (\lambda x.\ t[x])(\varepsilon(\lambda x.\ t[x])$ [`EQ_MP` 5,4]

7. $\vdash\ (\lambda x.\ t[x])v = t[v]$ [`BETA_CONV`]

8. $\vdash\ t[v] = (\lambda x.\ t[x])v$ [`SYM` 7]

9. $\Gamma_2,\ t[v]\ \vdash\ t'$ [Hypothesis]

10. $\Gamma_2\ \vdash\ t[v] \Rightarrow t'$ [`DISCH` 9]

11. $\Gamma_2\ \vdash\ (\lambda x.\ t[x])v \Rightarrow t'$ [`SUBST` 8,10]

12. $\Gamma_2,\ (\lambda x.\ t[x])v\ \vdash\ t'$ [`UNDISCH` 11]

13. $\Gamma_1 \cup \Gamma_2\ \vdash\ t'$ [`SELECT_ELIM` 6,12]

A.21 Use of a definition [`RIGHT_BETA_AP`]

$$\frac{\Gamma\ \vdash\ t = \lambda x_1 \cdots x_n.\ t'[x_1, \ldots, x_n]}{\Gamma\ \vdash\ t\ t_1 \cdots t_n = t'[t_1, \ldots, t_n]}$$

- Where none of the t_i contain any of the x_i.

1. $\Gamma\ \vdash\ t = \lambda x_1 \cdots x_n.\ t'[x_1, \ldots, x_n]$ [Suitably type-instantiated hypothesis]

2. $\Gamma\ \vdash\ t\ t_1 \cdots t_n = (\lambda x_1 \cdots x_n.\ t'[x_1, \ldots, x_n])\ t_1 \cdots t_n$ [`AP_THM` 1 (n times)]

3. $\vdash\ (\lambda x_1 \cdots x_n.\ t'[x_1, \ldots, x_n])\ t_1 \cdots t_n = t'[t_1, \ldots, t_n]$ [`BETA_CONV` (n times)]

4. $\Gamma\ \vdash\ t\ t_1 \cdots t_n = t'[t_1, \ldots, t_n]$ [`TRANS` 2,3]

A.22 $\wedge$-introduction [`CONJ`]

$$\frac{\Gamma_1\ \vdash\ t_1 \qquad\qquad \Gamma_2\ \vdash\ t_2}{\Gamma_1 \cup \Gamma_2\ \vdash\ t_1\ \wedge\ t_2}$$

1. $\vdash\ \wedge\ = \lambda b_1\ b_2.\ \forall b.\ (b_1 \Rightarrow (b_2 \Rightarrow b)) \Rightarrow b$ [Definition of $\wedge$]

2. $\vdash\ t_1\ \wedge\ t_2 = \forall b.\ (t_1 \Rightarrow (t_2 \Rightarrow b)) \Rightarrow b$ [`RIGHT_BETA_AP` 1]

3. $t_1 \Rightarrow (t_2 \Rightarrow b)\ \vdash\ t_1 \Rightarrow (t_2 \Rightarrow b)$ [`ASSUME`]

4. $\Gamma_1\ \vdash\ t_1$ [Hypothesis]

5. $\Gamma_1,\ t_1 \Rightarrow (t_2 \Rightarrow b)\ \vdash\ t_2 \Rightarrow b$ [`MP` 3,4]

6. $\Gamma_2\ \vdash\ t_2$ [Hypothesis]

7. $\Gamma_1 \cup \Gamma_2,\ t_1 \Rightarrow (t_2 \Rightarrow b)\ \vdash\ b$ [`MP` 5,6]

8. $\Gamma_1 \cup \Gamma_2 \vdash (t_1 \Rightarrow (t_2 \Rightarrow b)) \Rightarrow b$ [`DISCH` 7]

9. $\Gamma_1 \cup \Gamma_2 \vdash \forall b.\ (t_1 \Rightarrow (t_2 \Rightarrow b)) \Rightarrow b$ [`GEN` 8]

10. $\Gamma_1 \cup \Gamma_2 \vdash t_1 \wedge t_2$ [`EQ_MP` (`SYM` 2),9]

A.23 $\wedge$-elimination [`CONJUNCT1`, `CONJUNCT2`]

$$\frac{\Gamma \vdash t_1 \wedge t_2}{\Gamma \vdash t_1 \qquad\qquad \Gamma \vdash t_2}$$

1. $\vdash \wedge = \lambda b_1\ b_2.\ \forall b.\ (b_1 \Rightarrow (b_2 \Rightarrow b)) \Rightarrow b$ [Definition of $\wedge$]

2. $\vdash t_1 \wedge t_2 = \forall b.\ (t_1 \Rightarrow (t_2 \Rightarrow b)) \Rightarrow b$ [`RIGHT_BETA_AP` 1]

3. $\Gamma \vdash t_1 \wedge t_2$ [Hypothesis]

4. $\Gamma \vdash \forall b.\ (t_1 \Rightarrow (t_2 \Rightarrow b)) \Rightarrow b$ [`EQ_MP` 2,3]

5. $\Gamma \vdash (t_1 \Rightarrow (t_2 \Rightarrow t_1)) \Rightarrow t_1$ [`SPEC` 4]

6. $t_1 \vdash t_1$ [`ASSUME`]

7. $t_1 \vdash t_2 \Rightarrow t_1$ [`DISCH` 6]

8. $\vdash t_1 \Rightarrow (t_2 \Rightarrow t_1)$ [`DISCH` 7]

9. $\Gamma \vdash t_1$ [`MP` 5,8]

10. $\Gamma \vdash (t_1 \Rightarrow (t_2 \Rightarrow t_2)) \Rightarrow t_2$ [`SPEC` 4]

11. $t_2 \vdash t_2$ [`ASSUME`]

12. $\vdash t_2 \Rightarrow t_2$ [`DISCH` 11]

13. $\vdash t_1 \Rightarrow (t_2 \Rightarrow t_2)$ [`DISCH` 12]

14. $\Gamma \vdash t_2$ [`MP` 10,13]

15. $\Gamma \vdash t_1$ and $\Gamma \vdash t_2$ [9,14]

A.24 Right $\vee$-introduction [DISJ1]

$$\frac{\Gamma \vdash t_1}{\Gamma \vdash t_1 \vee t_2}$$

1. $\vdash \ \vee = \lambda b_1\ b_2.\ \forall b.\ (b_1 \Rightarrow b) \Rightarrow (b_2 \Rightarrow b) \Rightarrow b$ [Definition of $\vee$]
2. $\vdash t_1 \vee t_2 = \forall b.\ (t_1 \Rightarrow b) \Rightarrow (t_2 \Rightarrow b) \Rightarrow b$ [RIGHT_BETA_AP 1]
3. $\Gamma \vdash t_1$ [Hypothesis]
4. $t_1 \Rightarrow b \vdash t_1 \Rightarrow b$ [ASSUME]
5. $\Gamma,\ t_1 \Rightarrow b \vdash b$ [MP 4,3]
6. $\Gamma,\ t_1 \Rightarrow b \vdash (t_2 \Rightarrow b) \Rightarrow b$ [DISCH 5]
7. $\Gamma \vdash (t_1 \Rightarrow b) \Rightarrow (t_2 \Rightarrow b) \Rightarrow b$ [DISCH 6]
8. $\Gamma \vdash \forall b.\ (t_1 \Rightarrow b) \Rightarrow (t_2 \Rightarrow b) \Rightarrow b$ [GEN 7]
9. $\Gamma \vdash t_1 \vee t_2$ [EQ_MP (SYM 2),8]

A.25 Left $\vee$-introduction [DISJ2]

$$\frac{\Gamma \vdash t_2}{\Gamma \vdash t_1 \vee t_2}$$

1. $\vdash \ \vee = \lambda b_1\ b_2.\ \forall b.\ (b_1 \Rightarrow b) \Rightarrow (b_2 \Rightarrow b) \Rightarrow b$ [Definition of $\vee$]
2. $\vdash t_1 \vee t_2 = \forall b.\ (t_1 \Rightarrow b) \Rightarrow (t_2 \Rightarrow b) \Rightarrow b$ [RIGHT_BETA_AP 1]
3. $\Gamma \vdash t_2$ [Hypothesis]
4. $t_2 \Rightarrow b \vdash t_2 \Rightarrow b$ [ASSUME]
5. $\Gamma,\ t_2 \Rightarrow b \vdash b$ [MP 4,3]
6. $\Gamma \vdash (t_2 \Rightarrow b) \Rightarrow b$ [DISCH 5]
7. $\Gamma \vdash (t_1 \Rightarrow b) \Rightarrow (t_2 \Rightarrow b) \Rightarrow b$ [DISCH 6]
8. $\Gamma \vdash \forall b.\ (t_1 \Rightarrow b) \Rightarrow (t_2 \Rightarrow b) \Rightarrow b$ [GEN 7]
9. $\Gamma \vdash t_1 \vee t_2$ [EQ_MP (SYM 2),8]

A.26 $\vee$-elimination [DISJ_CASES]

$$\frac{\Gamma \vdash t_1 \vee t_2 \qquad \Gamma_1,\ t_1 \vdash t \qquad \Gamma_2,\ t_2 \vdash t}{\Gamma \cup \Gamma_1 \cup \Gamma_2 \vdash t}$$

1. $\vdash \vee = \lambda b_1\ b_2.\ \forall b.\ (b_1 \Rightarrow b) \Rightarrow (b_2 \Rightarrow b) \Rightarrow b$ [Definition of $\vee$]
2. $\vdash t_1 \vee t_2 = \forall b.\ (t_1 \Rightarrow b) \Rightarrow (t_2 \Rightarrow b) \Rightarrow b$ [RIGHT_BETA_AP 1]
3. $\Gamma \vdash t_1 \vee t_2$ [Hypothesis]
4. $\Gamma \vdash \forall b.\ (t_1 \Rightarrow b) \Rightarrow (t_2 \Rightarrow b) \Rightarrow b$ [EQ_MP 2,3]
5. $\Gamma \vdash (t_1 \Rightarrow t) \Rightarrow (t_2 \Rightarrow t) \Rightarrow t$ [SPEC 4]
6. $\Gamma_1,\ t_1 \vdash t$ [Hypothesis]
7. $\Gamma_1 \vdash t_1 \Rightarrow t$ [DISCH 6]
8. $\Gamma \cup \Gamma_1 \vdash (t_2 \Rightarrow t) \Rightarrow t$ [MP 5,7]
9. $\Gamma_2,\ t_2 \vdash t$ [Hypothesis]
10. $\Gamma_2 \vdash t_2 \Rightarrow t$ [DISCH 9]
11. $\Gamma \cup \Gamma_1 \cup \Gamma_2 \vdash t$ [MP 8,10]

A.27 Classical contradiction rule [CCONTR]

$$\frac{\Gamma,\ \neg t \vdash \mathsf{F}}{\Gamma \vdash t}$$

1. $\vdash \neg = \lambda b.\ b \Rightarrow \mathsf{F}$ [Definition of $\neg$]
2. $\vdash \neg t = t \Rightarrow \mathsf{F}$ [RIGHT_BETA_AP 1]
3. $\Gamma,\ \neg t \vdash \mathsf{F}$ [Hypothesis]
4. $\Gamma \vdash \neg t \Rightarrow \mathsf{F}$ [DISCH 3]
5. $\Gamma \vdash (t \Rightarrow \mathsf{F}) \Rightarrow \mathsf{F}$ [SUBST 2,4]
6. $t = \mathsf{F} \vdash t = \mathsf{F}$ [ASSUME]
7. $\Gamma,\ t = \mathsf{F} \vdash (\mathsf{F} \Rightarrow \mathsf{F}) \Rightarrow \mathsf{F}$ [SUBST 6,5]

8. $\mathsf{F} \ \vdash \ \mathsf{F}$ [`ASSUME`]

9. $\vdash \ \mathsf{F} \Rightarrow \mathsf{F}$ [`DISCH` 8]

10. $\Gamma, \ t = \mathsf{F} \ \vdash \ \mathsf{F}$ [`MP` 7,9]

11. $\vdash \ \mathsf{F} = \forall b.\ b$ [Definition of F]

12. $\Gamma, \ t = \mathsf{F} \ \vdash \ \forall b.\ b$ [`SUBST` 11,10]

13. $\Gamma, \ t = \mathsf{F} \ \vdash \ t$ [`SPEC` 12]

14. $\vdash \ \forall b.\ (b = \mathsf{T}) \ \vee \ (b = \mathsf{F})$ [Axiom]

15. $\vdash \ (t = \mathsf{T}) \ \vee \ (t = \mathsf{F})$ [`SPEC` 14]

16. $t = \mathsf{T} \ \vdash \ t = \mathsf{T}$ [`ASSUME`]

17. $t = \mathsf{T} \ \vdash \ t$ [`EQT_ELIM` 16]

18. $\Gamma \ \vdash \ t$ [`DISJ_CASES` 15,17,13]

B Predefined Theories

We describe below how the various non-primitive theories in HOL can be defined.

B.1 The theory prod

To define pairs in HOL we introduce a new theory `prod` with parent `bool` which contains the definition of the binary type operator *prod*. Values of type $(\sigma_1, \sigma_2)prod$ represent pairs whose first component has type σ_1 and whose second component has type σ_2. We will define an infix "," of type $\alpha \to \beta \to (\alpha, \beta)prod$ such that if $t_1{:}\sigma_1$ and $t_2{:}\sigma_2$ then (t_1, t_2) is a pair with first component t_1 and second component t_2.

The pair $(t_1{:}\sigma_1,\ t_2{:}\sigma_2)$ will be represented by the function $\lambda x\ y.\ (x = t_1) \wedge (y = t_2)$ which has type $\sigma_1 \to \sigma_2 \to bool$. We thus define $(\alpha, \beta)prod$ to be isomorphic to the subset of $\alpha \to \beta \to bool$ consisting of those functions f which have the property that $\vdash\ \exists a\ b.\ f = \lambda x\ y.\ (x = a) \wedge (y = b)$.

If we define the constants $\mathsf{Mk\_Pair}$ and $\mathsf{Is\_Pair}$ by:

$$\vdash\ \mathsf{Mk\_Pair}\ =\ \lambda a\ b.\ \lambda x\ y.\ (x = a) \wedge (y = b)$$
$$\vdash\ \mathsf{Is\_Pair}\ =\ \lambda f.\ \exists a\ b.\ f = \mathsf{Mk\_Pair}\ a\ b$$

then we can formally define the type $(\alpha, \beta)prod$ as follows:

1. The representing type is $\alpha \to \beta \to bool$.
2. The subset predicate is $\mathsf{Is\_Pair}$.
3. $\vdash\ \exists f.\ \mathsf{Is\_Pair}\ f$ because $\vdash\ \mathsf{Is\_Pair}(\lambda x\ y.\ (x = \varepsilon x'{:}\alpha.\ \mathsf{T}) \wedge (y = \varepsilon y'{:}\beta.\ \mathsf{T}))$
4. The representation function is $\mathsf{Rep\_Pair}{:}(\alpha, \beta)prod \to (\alpha \to \beta \to bool)$

Making this type definition introduces the axiom:

$$\vdash\ (\mathsf{One\_One\ Rep\_Pair})\ \wedge\ (\mathsf{Onto\_Subset\ Rep\_Pair\ Is\_Pair})$$

Types of the form $(\sigma_1, \sigma_2)prod$ will henceforth be written as $\sigma_1 \times \sigma_2$. If we define:

$$\vdash\ \mathsf{Abs\_Pair}\ =\ \mathsf{Inv\ Rep\_Pair}$$

then we can define the usual pairing operations by:

$$\begin{array}{lll} \vdash \ \$, & = & \lambda x{:}\alpha.\ \lambda y{:}\beta.\ \mathsf{Abs\_Pair}(\mathsf{Mk\_Pair}\ x\ y) \qquad\qquad (\text{, is an infix}) \\ \vdash \ \mathsf{Fst} & = & \lambda p{:}\alpha{\times}\beta.\ \varepsilon x.\ \exists y.\ p = (x, y) \\ \vdash \ \mathsf{Snd} & = & \lambda p{:}\alpha{\times}\beta.\ \varepsilon y.\ \exists x.\ p = (x, y) \end{array}$$

It follows from these definitions that:

$$\vdash \ \forall x\ y.\ \mathsf{Fst}(x, y) = x$$
$$\vdash \ \forall x\ y.\ \mathsf{Snd}(x, y) = y$$
$$\vdash \ \forall p{:}\alpha{\times}\beta.\ p = (\mathsf{Fst}\ p,\ \mathsf{Snd}\ p)$$

B.2 The theory num

We now sketch out how numbers can be defined. The idea is that *num* will be represented by the subset of *ind* consisting of Zero_Rep and all elements of the form $\mathsf{Suc\_Rep}^n$ Zero_Rep. It would be nice if we could simply define:

$$\vdash \ \mathsf{Is\_Num} \ = \ \lambda x.\ (x = \mathsf{Zero\_Rep}) \ \vee \ \exists n.\ x = \mathsf{Suc\_Rep}^n\ \mathsf{Zero\_Rep}$$

but we can't because $\mathsf{Suc\_Rep}^n$ Zero_Rep isn't a term (and even if it were the superscript n presupposes numbers have already been defined). The trick we use is the following:

$$\vdash \ \mathsf{Is\_Num} \ = \ \lambda x.\ \forall P.\ (P\ \mathsf{Zero\_Rep}) \ \wedge \ (\forall y.\ P\ y \Rightarrow P(\mathsf{Suc\_Rep}\ y)) \Rightarrow P x$$

It is straightforward to show from this definition that:

$$\vdash \ \mathsf{Is\_Num}\ \mathsf{Zero\_Rep}$$
$$\vdash \ \forall x.\ \mathsf{Is\_Num}\ x \Rightarrow \mathsf{Is\_Num}(\mathsf{Suc\_Rep}\ x)$$

We can now define the type *num* as follows:

1. The representing type is *ind*.
2. The subset predicate is Is_Num.
3. $\vdash\ \exists x.$ Is_Num x because $\vdash$ Is_Num Zero_Rep.
4. The representation function is Rep_Num:$num \to ind$.

To show that the type *num* defined this way is in fact the type of numbers we outline how Peano's postulates can be proved as theorems. These postulates are:

- There is a number 0.
- There is a function Suc called the successor function such that if n is a number then $\mathsf{Suc}\ n$ is a number.
- 0 is not the successor of any number.
- If two numbers have the same successor then they are equal.
- If a property holds of 0 and if whenever it holds of a number then it also holds of the successor of the number, then the property holds of all numbers. This postulate is called *Mathematical Induction*.

To define 0 and the successor function Suc it is useful to first define the inverse to the representation function $\mathsf{Rep\_Num}$.

$$\vdash\ \mathsf{Abs\_Num}\ =\ \mathsf{Inv\ Rep\_Num}$$

We can then define:

$$\vdash\ 0\ =\ \mathsf{Abs\_Num\ Zero\_Rep}$$
$$\vdash\ \mathsf{Suc}\ =\ \mathsf{Abs\_Num\ o\ Suc\_Rep\ o\ Rep\_Num}$$

Peano's postulates follow from these definitions. We will only sketch the proof of this. The first two postulates hold because $0{:}num$ and $\mathsf{Suc}{:}num{\rightarrow}num$. Because we chose $\mathsf{Zero\_Rep}$ not to be in the range of $\mathsf{Suc\_Rep}$ we can prove the following theorem which formalizes the third postulate:

$$\vdash\ \forall m.\ \neg(\mathsf{Suc}\ m = 0)$$

Because $\mathsf{Suc\_Rep}$ is one-to-one we can prove the following formalization of the forth postulate:

$$\vdash\ \forall m\ n.\ (\mathsf{Suc}\ m = \mathsf{Suc}\ n) \Rightarrow (m = n)$$

The fifth postulate, Mathematical Induction, follows from the definition of $\mathsf{Is\_Num}$.

$$\vdash\ \forall P{:}num{\rightarrow}bool.\ P\ 0\ \wedge\ (\forall m.\ P\ m \Rightarrow P(\mathsf{Suc}\ m)) \Rightarrow \forall m.\ P\ m$$

The numerals 1, 2, 3 etc. are defined by:

$$\begin{aligned}\vdash\ 1 &= \mathsf{Suc}\ 0\\ \vdash\ 2 &= \mathsf{Suc}(\mathsf{Suc}\ 0)\\ \vdash\ 3 &= \mathsf{Suc}(\mathsf{Suc}(\mathsf{Suc}\ 0))\\ &\bullet\\ &\bullet\\ &\bullet\end{aligned}$$

Because Suc is one-to-one these denote an infinite set of distinct values of type *num*.

B.3 The theory prim_rec

The usual theorems of arithmetic can be derived from Peano's postulates. The first step in doing this is to provide a mechanism for defining functions recursively. For example, the usual 'definition' of + is:

$$\begin{aligned}\vdash\ 0+m &= m\\ \vdash\ (\mathsf{Suc}\ m)+n &= \mathsf{Suc}(m+n)\end{aligned}$$

Unfortunately this isn't a definition. In order to convert such recursion equations into definitions we need the Primitive Recursion Theorem:

$$\begin{aligned}\vdash\ & \forall x{:}\alpha.\ \forall f{:}\alpha\to num\to\alpha.\ \exists fun{:}num\to\alpha.\\ & \quad (fun\ 0 = x)\ \wedge\\ & \quad (\forall m.\ fun(\mathsf{Suc}\ m) = f(fun\ m)m)\end{aligned}$$

The proof of this theorem from Peano's postulates is well known and was straightforward to do in the HOL system. As the details are fairly tricky (and boring) we have relegated them to Appendix C. To show that the Primitive Recursion Theorem solves the problem of defining + one specializes it by taking x to be $\lambda n.\ n$ and f to be $\lambda f'\ x'.\ \lambda n.\ \mathsf{Suc}(f'\ n)$, this yields:

$$\begin{aligned}\vdash\ \exists fun.\ & (fun\ 0 = (\lambda n.\ n))\ \wedge\\ & (\forall m.\ fun(\mathsf{Suc}\ m) = (\lambda f'\ x'.\ \lambda n.\ \mathsf{Suc}(f'\ n))\ (fun\ m)\ m)\end{aligned}$$

which is equivalent to:

$$\begin{aligned}\vdash\ \exists fun.\ & (fun\ 0\ n = n)\ \wedge\\ & (fun(\mathsf{Suc}\ m)n = \mathsf{Suc}(fun\ m\ n))\end{aligned}$$

Thus, if we define + by:

$$\vdash\ + \ = \ \varepsilon fun.\ \forall m\ n.\ (fun\ 0\ n = n) \wedge$$
$$(fun(\mathsf{Suc}\ m)n = \mathsf{Suc}(fun\ m\ n))$$

then it follows from the axiom for the ε-operator that:

$$\vdash\ 0 + n \ = \ n$$
$$\vdash\ (\mathsf{Suc}\ m) + n \ = \ \mathsf{Suc}(m + n)$$

as desired.

The method just used to define + generalizes to any primitive recursive definition. Such a definition has the form:

$$fun\ 0\ x_1\ \cdots\ x_n \ = \ f_1\ x_1\ \cdots\ x_n$$
$$fun\ (\mathsf{Suc}\ m)\ x_1\ \cdots\ x_n \ = \ f_2\ (fun\ m\ x_1\ \cdots\ x_n)\ m\ x_1\ \cdots\ x_n$$

where fun is the function being defined and f_1 and f_2 are given functions. To define a fun satisfying these equations we first define:

$$\vdash\ \mathsf{Prim\_Rec} \ = \ \lambda x\ f.\ \varepsilon fun.\ (fun\ 0 = x) \wedge$$
$$(\forall m.\ fun(\mathsf{Suc}\ m) = f(fun\ m)m)$$

It then follows by the axiom for the ε-operator and the Primitive Recursion Theorem that:

$$\vdash\ \mathsf{Prim\_Rec}\ x\ f\ 0 \ = \ x$$
$$\vdash\ \mathsf{Prim\_Rec}\ x\ f\ (\mathsf{Suc}\ m) \ = \ f\ (\mathsf{Prim\_Rec}\ x\ f\ m)\ m$$

A function fun satisfying the primitive recursive equations above can thus be defined by:

$$\vdash\ fun \ = \ \mathsf{Prim\_Rec}\ f_1\ (\lambda f\ m\ x_1\ \cdots\ x_n.\ f_2\ (f\ x_1\ \cdots\ x_n)\ m\ x_1\ \cdots\ x_n)$$

An example of a primitive recursion in this form is the definition of +:

$$\vdash\ + \ = \ \mathsf{Prim\_Rec}\ (\lambda x_1.\ x_1)\ (\lambda f\ m\ x_1.\ \mathsf{Suc}(f\ x_1))$$

This can be expressed more compactly as:

$$\vdash\ + \ = \ \mathsf{Prim\_Rec}\ \mathsf{I}\ (\lambda f\ m.\ \mathsf{Suc}\ \mathsf{o}\ f)$$

The HOL system automatically converts primitive recursive equations into definitions using Prim_Rec, and then proves that the constant so defined satisfies its 'defining' equations.

B.4 The theory arithmetic

The theory `arithmetic`, which is a descendant of `prim_rec`, contains the definitions of standard arithmetic functions and relations. These include the primitive recursive infixes $+$, $-$ and $\times$ which are defined so that:

$$\begin{array}{l}
\vdash\ (0+n=n)\ \wedge\ ((\mathsf{Suc}\ m)+n=\mathsf{Suc}(m+n))\\
\vdash\ (0-n=0)\ \wedge\ ((\mathsf{Suc}\ m)-n=((m<n)\rightarrow 0\mid \mathsf{Suc}(m-n)))\\
\vdash\ (0\times n=0)\ \wedge\ ((\mathsf{Suc}\ m)\times n=(m\times n)+n)
\end{array}$$

The division function is an infix / defined by:

$$\vdash\ m/n\ =\ \varepsilon x.\ m=n\times x$$

This satisfies:

$$\exists x.\ m=n\times x\ \vdash\ m=n\times(m/n)$$

The arithmetic relation $<$ is defined in the theory `prim_rec` (see Appendix C), the other relations are defined in `arithmetic` by:

$$\begin{array}{l}
\vdash\ m>n\ =\ (n<m)\\
\vdash\ m\leq n\ =\ (m<n)\ \vee\ (m=n)\\
\vdash\ m\geq n\ =\ (m>n)\ \vee\ (m=n)
\end{array}$$

The HOL system has many built-in elementary consequences of these definitions, they are proved when the system is constructed from its source files.

B.5 The theory list

The theory `list` contains the definition of a unary type operator *list*. Values of type σ *list* are finite lists of values of type σ. The two standard list processing functions:

$$\begin{array}{l}
\mathsf{Nil} : \alpha\ \mathit{list}\\
\mathsf{Cons} : \alpha{\rightarrow}(\alpha\ \mathit{list}){\rightarrow}(\alpha\ \mathit{list})
\end{array}$$

are defined in `list` and are proven to satisfy the the following Primitive Recursion Theorem for lists:

$$\vdash\ \forall x{:}\beta.\ \forall f{:}\beta{\rightarrow}\alpha{\rightarrow}(\alpha\ \mathit{list}){\rightarrow}\beta.\ \exists!\mathit{fun}{:}(\alpha\ \mathit{list}){\rightarrow}\beta.$$

$$(fun\ \mathsf{Nil} = x)\ \wedge$$
$$(\forall h\ t.\ fun(\mathsf{Cons}\ h\ t) = f(fun\ t)h\ t)$$

This theorem, which follows from the definitions of Cons, Nil and the type operator *list*, serves as the only 'axiom' for lists. It can be used to define the additional list processing functions:

$$\mathsf{Hd} : (\alpha\ \mathit{list}) \rightarrow \alpha$$
$$\mathsf{Tl} : (\alpha\ \mathit{list}) \rightarrow (\alpha\ \mathit{list})$$
$$\mathsf{Null} : (\alpha\ \mathit{list}) \rightarrow bool$$

The definitions of these functions (which are given later) ensure that they have the usual properties, namely:

$$\vdash\ \mathsf{Null\ Nil}$$
$$\vdash\ \forall x\ l.\ \neg(\mathsf{Null}(\mathsf{Cons}\ x\ l))$$
$$\vdash\ \forall x\ l.\ \mathsf{Hd}(\mathsf{Cons}\ x\ l) = x$$
$$\vdash\ \forall x\ l.\ \mathsf{Tl}(\mathsf{Cons}\ x\ l) = l$$
$$\vdash\ \forall l.\ \neg(\mathsf{Null}\ l) \Rightarrow \mathsf{Cons}(\mathsf{Hd}\ l)(\mathsf{Tl}\ l) = l$$

In addition we want lists to have the following property, which is analogous to induction for numbers:

$$\vdash\ \forall P.\ (P\ \mathsf{Nil})\ \wedge\ (\forall l.\ (P\ l) \Rightarrow \forall x.\ P(\mathsf{Cons}\ x\ l)) \Rightarrow \forall l.\ P\ l$$

This property follows directly from the uniqueness part of the primitive recursion theorem for lists given above.

We allow the following alternative notation for lists: the empty list Nil can be written as [ ] and a list of the form $\mathsf{Cons}\ t_1(\mathsf{Cons}\ t_2 \cdots (\mathsf{Cons}\ t_n\ \mathsf{Nil}) \cdots)$ can be written as $[t_1; \cdots ; t_n]$.

We will represent lists of type σ *list* by pairs (f, n) where f is a function of type $num \rightarrow \sigma$, and n is a number giving the length of the list. The idea is that $[t_0; \cdots ; t_{n-1}]$ is represented by (f, n) where for $i<n$ we have $f(i) = t_i$. Thus the head of such a list will be $f(0)$ and the tail will be represented by $((\lambda n.\ f(n{+}1)), n{-}1)$. The cons-function is represented by Cons_Rep which is defined so that:

$$\vdash\ \mathsf{Cons\_Rep}\ x\ (f, n) = ((\lambda i.\ ((i = 0) \rightarrow x \mid f(i{-}1))),\ n{+}1)$$

In order that equality (i.e. =) has the right meaning on lists we require that if (f_1, n) and (f_2, n) represent lists and have the property that:

$$\vdash\ \forall i.\ (i < n) \Rightarrow (f_1\ i = f_2\ i)$$

then $(f_1, n) = (f_2, n)$. We thus define the subset predicate $\mathsf{Is\_List}$ by:

$$\vdash \ \mathsf{Is\_List}\ r = \exists f\ n.\ r = ((\lambda m.\ (m < n \to (f\ m) \mid \varepsilon x{:}\alpha.\mathsf{T})), n)$$

Thus if $\vdash \ \mathsf{Is\_List}(f{:}\sigma, n)$ then for $i \geq n$ it will be the case that $f\ i = \varepsilon x{:}\sigma.\mathsf{T}$ (we use the ε-operator to chose an arbitrary but fixed value). It follows from this definition that:

$$\begin{array}{l} \vdash \ \mathsf{Is\_List}(f_1, n)\ \wedge \\ \quad \mathsf{Is\_List}(f_2, n)\ \wedge \\ \quad (\forall i.\ (i < n) \Rightarrow (f_1\ i = f_2\ i)) \Rightarrow \\ \quad ((f_1, n) = (f_2, n)) \end{array}$$

The formal definition of α *list* is:

1. The representing type is $(num{\to}\alpha){\times}num$.
2. The subset predicate is $\mathsf{Is\_List}$.
3. $\vdash \ \exists p.\ \mathsf{Is\_List}\ p$ because $\vdash \ \mathsf{Is\_List}((\lambda i{:}num.\ \varepsilon x{:}\alpha.\mathsf{T}),\ 0)$.
4. The representation function is $\mathsf{Rep\_List}{:}\alpha\ list{\to}((num{\to}\alpha){\times}num)$.

In order to define Cons and Nil it is convenient to first define the inverse to $\mathsf{Rep\_List}$:

$$\vdash \ \mathsf{Abs\_List}\ = \mathsf{Inv}\ \mathsf{Rep\_List}$$

using this we can now define:

$$\begin{array}{lcl} \vdash \ \mathsf{Nil} & = & \mathsf{Abs\_List}((\lambda i{:}num.\ \varepsilon x{:}\alpha.\mathsf{T}), 0) \\ \vdash \ \mathsf{Cons} & = & \lambda x{:}\alpha\ l{:}\alpha\ list.\ \mathsf{Abs\_List}(\mathsf{Cons\_Rep}\ x\ (\mathsf{Rep\_List}\ l)) \end{array}$$

The primitive recursion theorem for lists follows from these definitions and the fact that $\mathsf{Abs\_List}$ and its inverse, $\mathsf{Rep\_List}$, are isomorphisms between α *list* and the subset of $(num{\to}\alpha){\times}num$ for which $\mathsf{Is\_List}$ is true.

Using the primitive recursion theorem for lists, we can define $\mathsf{List\_Prim\_Rec}$ such that:

$$\vdash \ \mathsf{List\_Prim\_Rec}\ x\ f\ \mathsf{Nil}\ = \ x$$

$$\vdash \ \mathsf{List\_Prim\_Rec}\ x\ f\ (\mathsf{Cons}\ h\ t)\ =\ f\ (\mathsf{List\_Prim\_Rec}\ x\ f\ t)\ h\ t$$

If we define:

$$\begin{aligned}
&\vdash\ \mathsf{Hd}\ =\ \mathsf{List_Prim_Rec}(\lambda xyz.\ y)(\varepsilon x{:}\alpha.\mathsf{T})\\
&\vdash\ \mathsf{Tl}\ =\ \mathsf{List_Prim_Rec}(\lambda xyz.\ z)(\varepsilon l{:}(\alpha\ \mathit{list}).\mathsf{T})\\
&\vdash\ \mathsf{Null}\ =\ \mathsf{List_Prim_Rec}(\lambda xyz.\ \mathsf{F})\mathsf{T}
\end{aligned}$$

Then properties given above for these constants follow directly from the definition of List_Prim_Rec and the axiom for lists.

C The Primitive Recursion Theorem

In this appendix we outline a proof of the Primitive Recursion Theorem. The goal is to prove:

$$\begin{array}{l}\vdash\ \forall x{:}\alpha.\ \forall f{:}\alpha{\to}num{\to}\alpha.\ \exists fun{:}num{\to}\alpha.\\ \qquad (fun\ 0 = x) \wedge \\ \qquad (\forall m.\ fun(\mathsf{Suc}\ m) = f(fun\ m)m)\end{array}$$

It turns out to be sufficient to prove a slightly weaker result called the Simple Recursion Theorem, this is:

$$\begin{array}{l}\vdash\ \forall x{:}\alpha.\ \forall f{:}\alpha{\to}\alpha.\ \exists fun{:}num{\to}\alpha.\\ \qquad (fun\ 0 = x) \wedge \\ \qquad (\forall m.\ fun(\mathsf{Suc}\ m) = f(fun\ m))\end{array}$$

To establish this we explicitly construct the function fun from x and f by defining a constant Simp_Rec that can be proved to satisfy:

$$\begin{array}{l}\vdash\ \forall m\ x\ f.\ (\mathsf{Simp\_Rec}\ x\ f\ 0 = x) \wedge \\ \qquad\qquad (\mathsf{Simp\_Rec}\ x\ f\ (\mathsf{Suc}\ m) = f(\mathsf{Simp\_Rec}\ x\ f\ m))\end{array}$$

Defining Simp_Rec and showing it has this property is the hard part of the proof.

Here now is the actual sequence of theorems generated using the HOL system. Note that, unlike TPS [Andrews] or the Boyer-Moore theorem prover [Boyer-Moore], the HOL system is *not* fully automatic. To generate the theorems that follow the user has to tell the system how to do the proofs. The language used for giving this advice is ML [Gordon78].

To define Simp_Rec we need to use the less-than relation $<$. The natural definition of $<$ is by primitive recursion, however until we have shown that such recursive definitions are sound we cannot use them. The following non-recursive definition of $<$ works (note that it is higher order).

$$\vdash\ m < n\ =\ \exists P.\ (\forall i.\ P(\mathsf{Suc}\ i) \Rightarrow P\ i)\ \wedge\ P\ m\ \wedge\ \neg(P\ n)$$

From this it is routine to use Peano's postulates to deduce the following elementary lemmas about $<$. These lemmas are not intrinsically interesting, they are just the ones needed to fill in the details of the proof of the Primitive Recursion Theorem that I omit below. I list them here as they might be helpful to some enthusiastic reader who wants to generate these omitted details as an exercise.

$$\begin{array}{l}
\vdash \ \forall m\ n.\ (\mathsf{Suc}\ m = \mathsf{Suc}\ n) = (m = n) \\
\vdash \ \forall n.\ \neg(n < n) \\
\vdash \ \forall m\ n.\ (\mathsf{Suc}\ m) < n \Rightarrow m < n \\
\vdash \ \forall n.\ \neg(n < 0) \\
\vdash \ 0 < (\mathsf{Suc}\ 0) \\
\vdash \ \forall m\ n.\ m < n \Rightarrow (\mathsf{Suc}\ m) < (\mathsf{Suc}\ n) \\
\vdash \ \forall n.\ n < (\mathsf{Suc}\ n) \\
\vdash \ \forall m\ n.\ m < n \Rightarrow m < (\mathsf{Suc}\ n) \\
\vdash \ \forall m\ n.\ m < (\mathsf{Suc}\ n) \Rightarrow (m = n)\ \vee\ (m < n) \\
\vdash \ \forall m\ n.\ (m = n)\ \vee\ (m < n) \Rightarrow m < (\mathsf{Suc}\ n) \\
\vdash \ \forall m\ n.\ m < (\mathsf{Suc}\ n)\ =\ (m = n)\ \vee\ (m < n) \\
\vdash \ \forall m\ n.\ m < (\mathsf{Suc}\ n) \Rightarrow \neg(m = n) \Rightarrow (m < n) \\
\vdash \ \forall n.\ 0 < (\mathsf{Suc}\ n) \\
\vdash \ \forall n.\ (\mathsf{Suc}\ m = n) \Rightarrow m < n \\
\vdash \ \forall n.\ \neg(\mathsf{Suc}\ n = n) \\
\vdash \ \forall m\ n.\ (m = n) \Rightarrow \neg(m < n) \\
\vdash \ \forall m\ n.\ m < n \Rightarrow \neg(m = n)
\end{array}$$

In order to define $\mathsf{Simp\_Rec}$ we first define a relation $\mathsf{Simp\_Rec\_Rel}$, the idea is that $\mathsf{Simp\_Rec\_Rel}\ fun\ x\ f\ n$ is true if fun behaves like the function we are wanting to define (by simple recursion from x and f) for arguments less than n.

$$\begin{array}{l}
\vdash \ \mathsf{Simp_Rec_Rel}\ fun\ x\ f\ n\ = \\
\quad (fun\ 0 = x)\ \wedge\ (\forall m.\ m < n \Rightarrow (fun(\mathsf{Suc}\ m) = f(fun\ m)))
\end{array}$$

Using $\mathsf{Simp\_Rec\_Rel}$ we can define functions fun_0, fun_1, fun_2, ... such that fun_n satisfies the simple recursion equation for arguments less than n, i.e.:

$$\begin{array}{l}
\vdash \ \forall x\ f\ n.\ (fun_n\ x\ f\ 0 = x) \wedge \\
\qquad\qquad (\forall m.\ m < n \Rightarrow (fun_n\ x\ f\ (\mathsf{Suc}\ m) = f(fun_n\ x\ f\ m)))
\end{array}$$

A definition of fun_n that works is $fun_n = \mathsf{Simp\_Rec\_Fun}\ x\ f\ n$ where:

$$\vdash \ \mathsf{Simp\_Rec\_Fun}\ x\ f\ n\ =\ \varepsilon fun.\ \mathsf{Simp\_Rec\_Rel}\ fun\ x\ f\ n$$

Since fun_n (i.e. $\mathsf{Simp\_Rec\_Fun}\ x\ f\ n$) 'works' for all arguments less than n, and since n is always less than $n+1$, it follows that the function fun defined by $fun\ n = \lambda n.\ fun_{n+1}\ n$ works on all arguments. This fun is just $\mathsf{Simp\_Rec}\ x\ f$ where:

$$\vdash \ \mathsf{Simp\_Rec}\ x\ f\ n\ =\ \mathsf{Simp\_Rec\_Fun}\ x\ f\ (\mathsf{Suc}\ n)\ n$$

To formally verify the argument above we first use the definition of $\mathsf{Simp\_Rec\_Fun}$ and the property of the ε-operator to prove:

$$\begin{array}{l}\vdash\ (\exists fun.\ \mathsf{Simp\_Rec\_Rel}\ fun\ x\ f\ n)\ =\\ \quad ((\mathsf{Simp\_Rec\_Fun}\ x\ f\ n\ 0 = x)\ \wedge\\ \quad\ (\forall m.\ m < n \Rightarrow (\mathsf{Simp\_Rec\_Fun}\ x\ f\ n\ (\mathsf{Suc}\ m) =\\ \qquad\qquad\qquad\qquad f(\mathsf{Simp\_Rec\_Fun}\ x\ f\ n\ m))))\end{array}$$

By induction on n one can show:

$$\vdash\ \forall x\ f\ n.\ \exists fun.\ \mathsf{Simp\_Rec\_Rel}\ fun\ x\ f\ n$$

and hence:

$$\begin{array}{l}\vdash\ \forall x\ f\ n.\ (\mathsf{Simp\_Rec\_Fun}\ x\ f\ n\ 0 = x)\ \wedge\\ \qquad\qquad (\forall m.\ m < n \Rightarrow (\mathsf{Simp\_Rec\_Fun}\ x\ f\ n\ (\mathsf{Suc}\ m) =\\ \qquad\qquad\qquad\qquad f(\mathsf{Simp\_Rec\_Fun}\ x\ f\ n\ m)))\end{array}$$

This shows that the functions fun_0, fun_1, fun_2, $\ldots$ exist and have the necessary properties, namely:

$$\begin{array}{l}\vdash\ \forall x\ f\ n.\ fun_n\ x\ f\ 0\ =\ x\\ \vdash\ \forall m\ n.\ (m < n \Rightarrow (fun_n\ x\ f\ (\mathsf{Suc}\ m)\ =\ f(fun_n\ x\ f\ m)))\end{array}$$

Next we must show that if $n < m_1$ and $n < m_2$ then $fun_{m_1}\ n = fun_{m_2}\ n$. An induction on n yields:

$$\begin{array}{l}\vdash\ \forall n\ m_1\ m_2\ x\ f.\ n < m_1 \Rightarrow\\ \qquad\qquad\qquad n < m_2 \Rightarrow\\ \qquad\qquad\qquad (\mathsf{Simp\_Rec\_Fun}\ x\ f\ m_1\ n = \mathsf{Simp\_Rec\_Fun}\ x\ f\ m_2\ n)\end{array}$$

From the definition of $\mathsf{Simp\_Rec}$, and the following property of $<$:

$$\vdash\ \forall m.\ m < (\mathsf{Suc}\ m)\ \wedge\ m < (\mathsf{Suc}(\mathsf{Suc}\ m))$$

one can use the properties of fun_n to establish:

$$\begin{array}{l}\vdash\ \forall x\ f.\ (\mathsf{Simp\_Rec}\ x\ f\ 0 = x)\ \wedge\\ \qquad\quad (\forall m.\ \mathsf{Simp\_Rec}\ x\ f\ (\mathsf{Suc}\ m) = f(\mathsf{Simp\_Rec}\ x\ f\ m))\end{array}$$

The Simple Recursion Theorem follows directly from this.

To prove the full Primitive Recursion Theorem we define:

$$\vdash\ \mathsf{Prim\_Rec\_Fun}\ x\ f\ =\ \mathsf{Simp\_Rec}\ (\lambda n.\ x)\ (\lambda fun\ n.\ f(fun(\mathsf{Pre}\ n))n)$$

where Pre is defined by:

$$\vdash\ \mathsf{Pre}\ m\ =\ ((m = 0) \rightarrow 0 \mid \varepsilon n.\ m = \mathsf{Suc}\ n))$$

Using:

$$\vdash\ (\varepsilon n.\ m = n) = m$$

it is easy to show:

$$\vdash\ (\mathsf{Pre}\ 0 = 0)\ \wedge\ (\forall m.\ \mathsf{Pre}(\mathsf{Suc}\ m) = m)$$

We conclude the proof by defining:

$$\vdash\ \mathsf{Prim\_Rec}\ x\ f\ m\ =\ \mathsf{Prim\_Rec\_Fun}\ x\ f\ m\ (\mathsf{Pre}\ m)$$

and then using the Simple Recursion Theorem to prove:

$$\begin{aligned}\vdash\ \forall x\ f.\ & (\mathsf{Prim\_Rec}\ x\ f\ 0 = x) \wedge \\ & \quad (\forall m.\ \mathsf{Prim\_Rec}\ x\ f\ (\mathsf{Suc}\ m) = f(\mathsf{Prim\_Rec}\ x\ f\ m)m)\end{aligned}$$

The Primitive Recursion Theorem follows directly from this.

Q.E.D.