The HOL-Omega Logic

Peter V. Homeier

U. S. Department of Defense
palantir@trustworthytools.com
http://www.trustworthytools.com

Abstract. A new logic is posited for the widely used HOL theorem prover, as an extension of the existing higher order logic of the HOL4 system. The logic is extended to three levels, adding kinds to the existing levels of types and terms. New types include type operator variables and universal types as in System F. Impredicativity is avoided through the stratification of types by ranks according to the depth of universal types. The new system, called *HOL-Omega* or HOL_ω, is a merging of HOL4, HOL2P[11], and major aspects of System F_ω from chapter 30 of [10]. This document presents the abstract syntax and semantics for the kinds, types, and terms of the logic, as well as the new fundamental axioms and rules of inference. As the new logic is constructed according to the design principles of the LCF approach, the soundness of the entire system depends critically and solely on the soundness of this core.

1 Introduction

The HOL theorem prover [3] has had a wide influence in the field of mechanical theorem proving. Despite appearing in 1988 as one of the first tools in the field, HOL has enjoyed wide acceptance around the world, and continues to be used for many substantial projects, for example Anthony Fox's model of the ARM processor. HOL's influence is seen in that three other major theorem provers, HOL Light, ProofPower, and Isabelle/HOL, have used essentially the same logic.

One of the main reasons for HOL's influence has been that the actual logic implemented in the tool, higher order logic based on Church's simple theory of types, turns out to be both easy to work with and expressive enough to be able to support most models of hardware and software that people have wished to investigate. There are theorem provers with more powerful logics, and ones with less powerful logics, but it seems that classical higher order logic fortuitously found a "sweet-spot," balancing strong expressivity with nimble ease of use.

However, despite HOL's value, it has been recognized that there are some useful concepts beyond the power of higher order logic to state. An example is the practical device of monads. Monads are particularly useful in modelling, for example, realistic computations involving state or exceptions, as a shallow embedding in a logic which itself is strictly functional, without state or exceptions.

Individual monads can and have been expressed in HOL, and used to reduce the complexity of proofs about such real-world computations.

However, stating the general properties of all monads, and proving results about the class of all monads, has not been possible. The following shows why.

Let M be a postfix unary type operator that maps a type α to a type $\alpha\ M$, *unit* a prefix unary term operator of type $\alpha \to \alpha\ M$, and $\gg\!=$ an infix binary term operator of type $\alpha\ M \to (\alpha \to \beta\ M) \to \beta\ M$, where $k\ a \gg\!= h$ is $(k\ a) \gg\!= h$. Then M together with *unit* and $\gg\!=$ is a monad iff the following properties hold:

$$\begin{array}{lrcl} \text{left unit:} & \mathit{unit}\ a \gg\!= k & = & k\ a \\ \text{right unit:} & m \gg\!= \mathit{unit} & = & m \\ \text{associativity:} & m \gg\!= (\lambda a.\ k\ a \gg\!= h) & = & (m \gg\!= k) \gg\!= h \end{array}$$

There are two problems with this definition in higher order logic. First, while higher order logic includes type operator *constants* like `list` and `option`, it does not support type operator *variables* like M above.

But even if it did, consider the associativity property above. There are four occurrences of $\gg\!=$ in that property. Among these four instances are three distinct types. Unfortunately, in higher order logic, within a single expression a variable may only have a single type. So this property would not type-check.

This is annoying because if $\gg\!=$ were a *constant* instead of a variable, these different instances of its basic type would be supported. What we need is a way to give $\gg\!=$ a single type which can then be specialized for each of $\gg\!=$'s four instances to produce the three distinct types required.

One way is to introduce *universal types*, as in System F [10]. A universal type is written $\forall\alpha.\sigma$, where α is a type variable and σ is a type expression, possibly including α. Such occurrences of α are bound by the universal quantification.

In addition, System F introduces abstractions of types over terms, written as $\lambda{:}\alpha.t$, where α is a type variable and t is a term. This yields a term, whose type is a universal type. Specifically, if t has type σ, then $\lambda{:}\alpha.t$ has type $\forall\alpha.\sigma$.

Given such an abstraction t, it is specialized for a particular type by $t[{:}\sigma{:}]$. This gives rise to a new form of beta-reduction on term-type applications, where $(\lambda{:}\alpha.t)[{:}\sigma{:}]$ reduces to $t[\sigma/\alpha]$. For convenience, we write $t[{:}\alpha,\beta{:}]$ for $(t[{:}\alpha{:}])[{:}\beta{:}]$.

Given these new forms, we can express the types of *unit* and $\gg\!=$ as

$$\begin{array}{rl} \mathit{unit}: & \forall\alpha.\ \alpha \to \alpha\ M \\ \gg\!=: & \forall\alpha\ \beta.\ \alpha\ M \to (\alpha \to \beta\ M) \to \beta\ M \end{array}$$

and the three monad properties as

$$\begin{array}{rcl} \mathit{unit}[{:}\alpha{:}]\ a\ (\gg\!=[{:}\alpha,\beta{:}])\ k & = & k\ a \\ m\ (\gg\!=[{:}\alpha,\alpha{:}])\ (\mathit{unit}[{:}\alpha{:}]) & = & m \\ m\ (\gg\!=[{:}\alpha,\gamma{:}])\ (\lambda a.\ k\ a\ (\gg\!=[{:}\beta,\gamma{:}])\ h) & = & (m\ (\gg\!=[{:}\alpha,\beta{:}])\ k)\ (\gg\!=[{:}\beta,\gamma{:}])\ h \end{array}$$

What we have done here is take manual control of the typing. Since the normal HOL parametric polymorphism was inadequate, we have added facilities for type abstraction and instantiation of terms. This allows the single type of a variable to be specialized for different occurrences within the same expression.

Given the existing polymorphism in HOL, in practice universal types are needed only rarely; but when they are needed, they are absolutely essential.

In related work, as early as 1993 Tom Melham advocated adding quantification over type variables [8]. HOL-Omega includes such quantification, defining it using abstraction over type variables. Norbert Völker's HOL2P [11], a direct ancestor of this work, supports universal types quantifying over types of rank 0. HOL2P is approximately the same as HOL-Omega, but without kinds, curried type operators, or ranks > 1. Benjamin C. Pierce [10] describes a variety of programming languages with advanced type systems. HOL-Omega is similar to his system F_ω of chapter 30, but avoids F_ω's impredicativity. HOL-Omega does not include dependent types, such as found in the calculus of constructions.

In the remainder of this paper, we describe the core logic of the HOL-Omega system, and some additions to the core. In Section 2, we present the abstract syntax of HOL-Omega. Section 3 describes the set-theoretic semantics for the logic. Section 4 gives the new core rules of inference and axioms. Section 5 covers additional type and term definitions on top of the core. Section 6 presents a number of examples using the expanded logic, and in Section 7 we conclude.

2 Syntax of the HOL-Omega Logic

For reasons of space, we assume the reader is familiar with the types, terms, axioms, and rules of inference of the HOL logic, as described in [3,4,5,6]. This section presents the abstract syntax of the new HOL-Omega logic.

In HOL-Omega, the syntax consists of ranks, kinds, types, and terms.

2.1 Ranks

Ranks are natural numbers indicating the depth of universal type quantification present or permitted in a type. We use the variable r to range over ranks.

rank ::= natural

The purpose of ranks is to avoid impredicativity, which is inconsistent with HOL [2]. However, a naïve interpretation has been found to be too constrictive. For example, the HOL identity operator `I` has type $\alpha \to \alpha$, where α has rank 0. However, it is entirely natural to expect to apply `I` to values of higher ranks, and to expect `I` to function as the identity function on those higher-rank values. To have an infinite set of identity functions, one for each rank, would be absurd.

Inspired by new set theory, John Matthews suggested the idea of considering all ranks as being formed as a sum of a variable and a natural, where there is only one rank variable, z, ranging over naturals. This reflects the intuition that if a mathematical development was properly constructed at one rank, it could as easily have been constructed at the rank one higher, consistently at each step of the development. Only one rank variable is necessary to capture this intuition, representing the finite number of ranks that the entire development is promoted.

If there is only one rank variable, it may be understood to be always present without being explicitly modeled. Thus rank 2 signifies $z+2$. A rank substitution

θ_r indicates the single mapping $z \mapsto z + \theta_r$, so applying θ_r to $z + n$ yields $z + (n + \theta_r)$. A rank r' is an instance of r if $r' = r[\theta_r]$ for some θ_r, i.e., if $r' \geq r$.

2.2 Kinds

HOL-Omega introduces kinds as a new level in the logic, not present in HOL. Kinds control the proper formation of types just as types do for terms.

There are three varieties of kinds, namely the base kind (the kind of proper types), kind variables, and arrow kinds (the kinds of type operators).

$$\begin{array}{lll} \text{kind} ::= & \mathbf{ty} & \textit{(base kind)} \\ \quad\mid & \kappa & \textit{(kind variable)} \\ \quad\mid & k_1 \Rightarrow k_2 & \textit{(arrow kind)} \end{array}$$

We use the variable k to range over kinds, and κ to range over kind variables. The arrow kind $k_1 \Rightarrow k_2$ has domain k_1 and range k_2. Arrow kinds are also called *higher kinds*, meaning higher than the base kind. A kind k' is an instance of k if $k' = k[\theta_k]$ for some substitution θ_k, a mapping from kind variables to kinds.

2.3 Types

Replacing HOL's two varieties of types, HOL-Omega has five: type variables, type constants, type applications, type abstractions, and universal types.

type-variable ::= name × kind × rank
type-constant ::= name × kind × rank *(instance of kind in env.)*

$$\begin{array}{lll} \text{type} ::= & \alpha & \textit{(type-variable)} \\ \quad\mid & \tau & \textit{(type-constant)} \\ \quad\mid & \sigma_{arg}\ \sigma_{opr} & \textit{(type application, postfix syntax)} \\ \quad\mid & \lambda\alpha.\ \sigma & \textit{(type abstraction)} \\ \quad\mid & \forall\alpha.\ \sigma & \textit{(universal type)} \end{array}$$

We will use α to range over type variables, τ to range over type constants, and σ to range over types. Type constants must have kinds which are instances of the environment's kind for that type constant name.

Kinding: $\boxed{\sigma : k}$	$\dfrac{}{\alpha : \textit{kind of } \alpha}$ $\dfrac{}{\tau : \textit{kind of } \tau}$	$\dfrac{\sigma_{opr} : k_1 \Rightarrow k_2,\quad \sigma_{arg} : k_1}{\sigma_{arg}\ \sigma_{opr} : k_2}$	$\dfrac{\alpha : k_1,\quad \sigma : k_2}{\lambda\alpha.\sigma : k_1 \Rightarrow k_2}$ $\dfrac{\alpha : k,\quad \sigma : \mathbf{ty}}{\forall\alpha.\sigma : \mathbf{ty}}$
Ranking: $\boxed{\sigma :\leq r}$	$\dfrac{}{\alpha :\leq \textit{rank of } \alpha}$ $\dfrac{}{\tau :\leq \textit{rank of } \tau}$	$\dfrac{\sigma_{opr} :\leq r_2,\quad \sigma_{arg} :\leq r_1}{\sigma_{arg}\ \sigma_{opr} :\leq \max(r_1,r_2)}$ $\dfrac{\sigma :\leq r,\quad r \leq r'}{\sigma :\leq r'}$	$\dfrac{\alpha :\leq r_1,\quad \sigma :\leq r_2}{\lambda\alpha.\sigma :\leq \max(r_1,r_2)}$ $\dfrac{\alpha :\leq r_1,\quad \sigma :\leq r_2}{\forall\alpha.\sigma :\leq \max(r_1{+}1,r_2)}$
Typing: $\boxed{t : \sigma}$	$\dfrac{}{x : \textit{type of } x}$ $\dfrac{}{c : \textit{type of } c}$	$\dfrac{t_{opr} : \sigma_1 \rightarrow \sigma_2,\quad t_{arg} : \sigma_1}{t_{opr}\ t_{arg} : \sigma_2}$ $\dfrac{t : \forall\alpha{:}k{:}{\leq}r.\ \sigma',\quad \sigma : k :\leq r}{t\,[{:}\sigma{:}] : \sigma'[\sigma/\alpha]}$	$\dfrac{x : \sigma_1,\quad t : \sigma_2}{\lambda x.t : \sigma_1 \rightarrow \sigma_2}$ $\dfrac{\alpha : k :\leq r,\quad t : \sigma}{\lambda{:}\alpha.t : \forall\alpha.\sigma}$

Existing types of HOL are fully supported in HOL-Omega. HOL type variables are represented as HOL-Omega type variables of kind **ty** and rank 0. HOL type applications of a type constant to a list of type arguments are represented in HOL-Omega as a curried type constant applied to the arguments in sequence, as $(\alpha_1, ..., \alpha_n)\tau = \alpha_n\ (...\ (\alpha_1\ \tau)...)$.

We write $\sigma : k :\leq r$ to say that type σ has kind k and rank r.

Proper types are types of kind **ty**; only these types can be the type of a term.

In a type application of a type operator to an argument, the operator must have an arrow kind, and the domain of the kind of the operator must equal the kind of the argument. If so, the kind of the result of the type application will be the range of the kind of the operator. Also, the body of a universal type must have the base kind. These restrictions ensure types are well-kinded.

In both universal types and type abstractions, the type variable is bound over the type body. This binding structure introduces the notions of alpha and beta equivalence, as direct analogs of the corresponding notions for terms. In fact, types are identified up to alpha-beta equivalence. The following denote the same type: $\lambda\alpha.\alpha$, $\lambda\beta.\beta$, $\lambda\beta.\beta(\lambda\alpha.\alpha)$, $\gamma(\lambda\alpha.\lambda\beta.\beta)$. Beta reduction is of the form $\sigma_2(\lambda\alpha.\sigma_1) = \sigma_1[\sigma_2/\alpha]$, where $\sigma_1[\sigma_2/\alpha]$ is the result of substituting σ_2 for all free occurrences of α in σ_1, with bound type variables in σ_1 renamed as necessary.

A type σ' is an instance of σ if $\sigma'{=}\sigma[\theta_r][\theta_k][\theta_\sigma]$ for some rank, kind, and type substitutions $\theta_r \in \mathsf{N}$, θ_k mapping kind variables to kinds, and θ_σ mapping type variables to types. The substitutions are applied in sequence, with θ_r first.

When matching two types, the matching is higher order, so the pattern $\alpha \to \alpha\ \mu$ (where $\mu : \mathbf{ty} \Rightarrow \mathbf{ty}$) matches $\beta \to \beta$, yielding $[\alpha \mapsto \beta, \mu \mapsto \lambda\alpha.\alpha]$.

The primeval environment contains the type constants `bool`, `ind`, and `fun` as in HOL, where `bool` : **ty**, `ind` : **ty**, and `fun` : **ty** ⇒ **ty** ⇒ **ty**, and all three have rank 0. `fun` is usually written as the binary infix type operator →, and for a function type $\sigma_1 \to \sigma_2$, we say that the domain is σ_1 and the range is σ_2. Also, for a universal type $\forall\alpha.\sigma$, we say that the domain is α and the range is σ.

2.4 Terms

HOL-Omega adds to the existing four varieties of terms two new varieties, namely term-type applications and type-term abstractions. We use x to range over term variables, c over term constants, and t over terms.

variable ::= name × type
constant ::= name × type (*an instance of type stored in environment)*

term ::= x *(variable)*
| c *(constant)*
| $t_{opr}\ t_{arg}$ *(application, prefix syntax)*
| $\lambda x.\ t$ *(abstraction)*
| $t\ [:\sigma:]$ *(term-type application)*
| $\lambda{:}\alpha.\ t$ *(type-term abstraction)*

In applications $t_1\ t_2$, the domain of the type of t_1 must equal the type of t_2.

As in System F, in abstractions of a type variable over a term $\lambda{:}\alpha.t$, the type variable α must not occur freely in the type of any free variable of the term t.

There are three important restrictions on term-type applications (t $[{:}\,\sigma\,{:}]$).

1. The type of the term t must be a universal type, say $\forall\alpha.\sigma'$.
2. The kind of α must match the kind of the type argument σ.
3. The rank of α must contain ($\geq$) the rank of the type argument σ.

The first and second restrictions ensure terms are well-typed and well-kinded.

The third restriction is necessary to avoid impredicativity, for a simpler set-theoretic model. This restriction means that the type argument is validly one of the types over which the universal type quantifies. On this key restriction, the consistency of HOL-Omega rests.

3 Semantics of the HOL-Omega Logic

3.1 A Universe for HOL-Omega Kinds, Types, and Terms

We give the ZFC semantics of HOL-Omega kinds, types, and terms in terms of a universe $\mathcal{U}$, which is fixed set of sets of sets. This development draws heavily from Pitts[6] and Völker[11]. We construct $\mathcal{U}$ as a result of first constructing sequences of sets $\mathcal{U}_0,\mathcal{U}_1,\mathcal{U}_2,...$, and $\mathcal{T}_0,\mathcal{T}_1,\mathcal{T}_2,...$, where $\mathcal{U}_i$ and $\mathcal{T}_i$ will only involve types of rank $\leq i$. Kinds will be modeled as elements K of $\mathcal{U}$, types will be modeled as elements T of $K \in \mathcal{U}$, and terms will be modeled as elements E of $T \in \mathcal{T} \in \mathcal{U}$.

There exist $\mathcal{U}_i$ and $\mathcal{T}_i$ for $i = 0, 1, 2, ...$, satisfying the following properties:

Inhab. Each element of $\mathcal{U}_i$ is a non-empty set of non-empty sets.
Typ. $\mathcal{U}_i$ contains a distinguished element $\mathcal{T}_i$.
Arrow. If $K \in \mathcal{U}_i$ and $L \in \mathcal{U}_i$, then $K{\to}L \in \mathcal{U}_i$, where $X{\to}Y$ is the set-theoretic (total) function space from the set X to the set Y.
Clos. $\mathcal{U}_i$ has no elements except those by **Typ** or **Arrow**.
Ext. $\mathcal{T}_{i+1}$ extends $\mathcal{T}_i$: $\mathcal{T}_i \subseteq \mathcal{T}_{i+1}$.
Sub. If $X \in \mathcal{T}_i$ and $\emptyset \neq Y \subseteq X$, then $Y \in \mathcal{T}_i$.
Fun. If $X \in \mathcal{T}_i$ and $Y \in \mathcal{T}_i$, then $X{\to}Y \in \mathcal{T}_i$.
Univ. If $K \in \mathcal{U}_i$ and $f : K \to \mathcal{T}_{i+1}$, then $\prod_{X\in K} fX \in \mathcal{T}_{i+1}$. The set theoretic product $\prod_{X\in K} fX$ is the set of all functions $g : K \to \bigcup_{X\in K} fX$ such that for all $X \in K$, $gX \in fX$.
Bool. $\mathcal{T}_0$ contains a distinguished 2-element set $\mathrm{B} = \{\mathbf{true}, \mathbf{false}\}$.
Infty. $\mathcal{T}_0$ contains a distinguished infinite set I.
AllTyp. $\mathcal{T}$ is defined to be $\bigcup_{i\in\mathsf{N}} \mathcal{T}_i$.
AllArr. $\mathcal{U}$ is the closure of $\{\mathcal{T}\}$ under set theoretic function space creation.
Choice. There are distinguished elements $\mathrm{chty}_i \in \prod_{K\in\mathcal{U}_i} K$ and $\mathrm{ch} \in \prod_{X\in\mathcal{T}} X$. For all i and for all $K \in \mathcal{U}_i$, K is nonempty and $\mathrm{chty}_i(K) \in K$ is an example of this, and for all $X \in \mathcal{T}$, X is nonempty and $\mathrm{ch}(X) \in X$ is an example of this.

The system consisting of the above properties is consistent. The following construction is from William Schneeburger. Let $\mathcal{U}_i$ be the closure of $\{\mathcal{T}_i\}$ under

Arrow. Given $\mathcal{T}_i$ and $\mathcal{U}_i$, we can construct $\mathcal{T}_{i+1}$ by iteration over the ordinals [9]. Let $\mathcal{S}_0 = \mathcal{T}_i$. For all ordinals α, let $\mathcal{S}_{\alpha+1}$ be the closure under **Sub** and **Fun** of

$$\mathcal{S}_\alpha \bigcup \{ \prod_{X \in K} f\, X \mid K \in \mathcal{U}_i \ \wedge \ f : K \to \mathcal{S}_\alpha \}.$$

For limit ordinals λ, let $\mathcal{S}_\lambda = \bigcup_{\alpha<\lambda} \mathcal{S}_\alpha$, which is closed under **Sub** and **Fun**. Let $\mathfrak{n} = |\bigcup_{K \in \mathcal{U}_i} K|$. Then $|K| \leq \mathfrak{n}$ for all $K \in \mathcal{U}_i$. Let $\mathfrak{m} = \mathfrak{n}^+$, the least cardinal $> \mathfrak{n}$. Then $\mathfrak{m}$ is a regular cardinal [9, p. 146] $> |K|$ for all $K \in \mathcal{U}_i$. Then we define $\mathcal{T}_{i+1} = \mathcal{S}_\mathfrak{m}$, which is sufficiently large by the following theorem.

Theorem 1. *$\mathcal{S}_\mathfrak{m}$ is closed under* **Univ** *(as well as* **Sub** *and* **Fun***).*

Proof. Suppose $K \in \mathcal{U}_i$ and $f : K \to \mathcal{S}_\mathfrak{m}$. $\mathcal{S}_\mathfrak{m} = \bigcup_{\alpha<\mathfrak{m}} \mathcal{S}_\alpha$, so for each $X \in K$ define $\gamma_X =$ the smallest α s.t. $f\, X \in \mathcal{S}_\alpha$, thus $\gamma_X < \mathfrak{m}$. Define $\Gamma = \{\gamma_X | X \in K\}$. Then $\Gamma \subseteq \mathfrak{m}$, and $|K| < \mathfrak{m}$ so $|\Gamma| < \mathfrak{m}$ thus $\bigcup \Gamma < \mathfrak{m}$ since $\mathfrak{m}$ is regular. The image of $f \subseteq \mathcal{S}_{\bigcup \Gamma}$, so by the definition of $\mathcal{S}_{\alpha+1}$, $\prod_{X \in K} f\, X \in \mathcal{S}_{(\bigcup \Gamma)+1} \subseteq \mathcal{S}_\mathfrak{m}$. □

3.2 Constraining Kinds and Types to a Particular Rank

The function $\_\!\Downarrow\! r$ transforms an element K of $\mathcal{U}$ into an element of $\mathcal{U}_r$:

$$\mathcal{T}\!\Downarrow\! r = \mathcal{T}_r$$
$$(K_1 \to K_2)\!\Downarrow\! r = K_1\!\Downarrow\! r \to K_2\!\Downarrow\! r$$

We need to map some elements $T \in K \in \mathcal{U}$ down to the corresponding elements in $K\!\Downarrow\! r \in \mathcal{U}_r$, when T is consistent with a type of rank r. Not all T can be so mapped; we define the subset of K that can, and the mapping, as follows.

We define the subset $K|r \subseteq K \in \mathcal{U}$ as the elements consistent with rank r, and the function $\_\!\downarrow\! r$ which transforms an element T of $K|r$ into one of $K\!\Downarrow\! r$, mutually recursively on the structure of K:

$$\begin{aligned} \mathcal{T}|r &= \mathcal{T}_r \\ (K_1 \to K_2)|r &= \{f \mid f \in K_1 \to K_2 \ \wedge \\ &\qquad \forall (x,y) \in f.\ (x \in K_1|r \ \Rightarrow \ y \in K_2|r) \ \wedge \\ &\qquad f\!\downarrow\! r \text{ is a function}\} \end{aligned}$$

$$\begin{array}{lll} \text{If } T \in \mathcal{T}|r, & \text{then} & T\!\downarrow\! r = T \\ \text{If } T \in (K_1 \to K_2)|r, & \text{then} & T\!\downarrow\! r = \{(x\!\downarrow\! r,\ y\!\downarrow\! r) \mid (x,y) \in T \wedge x \in K_1|r\} \end{array}$$

If $K = K_1 \to K_2$, by the definition of $T\!\downarrow\! r$, $T\!\downarrow\! r \subseteq K_1\!\Downarrow\! r \times K_2\!\Downarrow\! r$, and by $T \in K|r$, $T\!\downarrow\! r$ is a function, so $T\!\downarrow\! r \in K_1\!\Downarrow\! r \to K_2\!\Downarrow\! r = (K_1 \to K_2)\!\Downarrow\! r = K\!\Downarrow\! r$.

We can define $\_\!\Uparrow\! r : \mathcal{U}_r \to \mathcal{U}$ and $\_\!\uparrow\! r : K\!\Downarrow\! r \to K|r$ as the inverses of $\_\!\Downarrow\! r$ and $\_\!\downarrow\! r$, so that $(K\!\Uparrow\! r)\!\Downarrow\! r = K$ for all $K \in \mathcal{U}_r$ and $(T\!\uparrow\! r)\!\downarrow\! r = T$ for all $T \in K \in \mathcal{U}_r$.

$$\mathcal{T}_r\!\Uparrow\! r = \mathcal{T}$$
$$(K_1 \to K_2)\!\Uparrow\! r = K_1\!\Uparrow\! r \to K_2\!\Uparrow\! r$$

$$\begin{array}{lll} \text{If } T \in \mathcal{T}\!\Downarrow\! r, & \text{then} & T\!\uparrow\! r = T \\ \text{If } T \in (K_1 \to K_2)\!\Downarrow\! r, & \text{then} & T\!\uparrow\! r = \lambda(x \in K_1).\ \text{if } x \in K_1|r \text{ then } (T(x\!\downarrow\! r))\!\uparrow\! r \\ & & \qquad\qquad\qquad\qquad\qquad\quad \text{else chtype}(K_2, r) \end{array}$$
$$\text{where chtype}(K, r) = (\text{chty}_r(K\!\Downarrow\! r))\!\uparrow\! r$$

3.3 Semantics of Ranks and Kinds

As mentioned earlier, ranks syntactically appear as natural numbers r, but are actually combined with the hidden single rank variable z as $z + r$. A rank environment $\zeta \in \mathsf{N}$ gives the value of z. The semantics of ranks is then $[\![r]\!]_\zeta = \zeta + r$.

A kind environment ξ is a mapping from kind variables to elements of $\mathcal{U}$.

The semantics of kinds $[\![k]\!]_\xi$ is defined by recursion over the structure of k:

$$\begin{aligned} [\![\mathbf{ty}]\!]_\xi &= \mathcal{T} \\ [\![\kappa]\!]_\xi &= \xi\ \kappa \\ [\![k_1 \Rightarrow k_2]\!]_\xi &= [\![k_1]\!]_\xi \to [\![k_2]\!]_\xi \end{aligned}$$

3.4 Semantics of Types

We will distinguish `bool`, `ind`, and function types $\sigma_1 \to \sigma_2$ as special cases, in order to ensure a standard model. We assume a model M that takes a rank and kind environment (ζ, ξ) and gives a valuation of each type constant τ of kind k and rank r as an element of $[\![k]\!]_\xi \mid [\![r]\!]_\zeta$. For clarity we omit the decoration $[\![\_]\!]_M$.

A type environment ρ takes a rank and a kind environment (ζ, ξ) to a mapping of each type variable α of kind k and rank r to a value $T \in [\![k]\!]_\xi \mid [\![r]\!]_\zeta$.

$[\![\sigma]\!]_{\zeta,\xi,\rho}$ is defined by recursion over the structure of σ:

$$\begin{aligned} [\![\texttt{bool}]\!]_{\zeta,\xi,\rho} &= \mathrm{B} \\ [\![\texttt{ind}]\!]_{\zeta,\xi,\rho} &= \mathrm{I} \\ [\![\, \sigma_1 \to \sigma_2 \,]\!]_{\zeta,\xi,\rho} &= [\![\sigma_1]\!]_{\zeta,\xi,\rho} \to [\![\sigma_2]\!]_{\zeta,\xi,\rho} \\ [\![\tau]\!]_{\zeta,\xi,\rho} &= M\ (\zeta,\xi)\ \tau \\ [\![\alpha]\!]_{\zeta,\xi,\rho} &= \rho\ (\zeta,\xi)\ \alpha \\ [\![\, \sigma_{arg}\ \sigma_{opr} \,]\!]_{\zeta,\xi,\rho} &= [\![\sigma_{opr}]\!]_{\zeta,\xi,\rho}\ [\![\sigma_{arg}]\!]_{\zeta,\xi,\rho} \\ [\![\, \lambda(\alpha : k :\leq r).\ \sigma \,]\!]_{\zeta,\xi,\rho} &= \lambda T \in [\![k]\!]_\xi . \begin{cases} [\![\sigma]\!]_{\zeta,\xi,\rho[\alpha \mapsto T]} & \text{if } T \in [\![k]\!]_\xi \mid [\![r]\!]_\zeta \\ \mathrm{chtype}([\![k_\sigma]\!]_\xi, [\![r_\sigma]\!]_\zeta) & \text{otherwise} \end{cases} \\ [\![\, \forall(\alpha : k :\leq r).\ \sigma \,]\!]_{\zeta,\xi,\rho} &= \textstyle\prod_{T \in [\![k]\!]_\xi \Downarrow [\![r]\!]_\zeta}\ [\![\sigma]\!]_{\zeta,\xi,\rho[\alpha \mapsto T \Uparrow [\![r]\!]_\zeta]} \end{aligned}$$

where for $[\![\, \lambda(\alpha : k :\leq r).\ \sigma \,]\!]_{\zeta,\xi,\rho}$, if T has rank larger than the variable α, an arbitrary type of the kind k_σ and rank r_σ of σ is returned, essentially as an error.

By induction over the structure of types, it can be demonstrated that the semantics of types is consistent with the semantics of kinds and ranks, i.e.,

$$[\![\, \sigma : k :\leq r \,]\!]_{\zeta,\xi,\rho} \in [\![k]\!]_\xi \mid [\![r]\!]_\zeta .$$

3.5 Semantics of Terms

In addition to the type mapping described above, the model M is assumed, given a triple of a rank, kind, and type environments, to provide a valuation of each term constant c of type σ as an element of $[\![\sigma]\!]_{\zeta,\xi,\rho}$. A term environment μ takes a triple of a rank, kind, and type environments to a mapping of each term variable x of type σ to a value v which is an element of $[\![\sigma]\!]_{\zeta,\xi,\rho}$.

$[\![\, t \,]\!]_{\zeta,\xi,\rho,\mu}$ is defined by recursion over the structure of t:

$$\begin{aligned}
[\![\, c \,]\!]_{\zeta,\xi,\rho,\mu} &= M\ (\zeta,\xi,\rho)\ c \\
[\![\, x \,]\!]_{\zeta,\xi,\rho,\mu} &= \mu\ (\zeta,\xi,\rho)\ x \\
[\![\, t_1\ t_2 \,]\!]_{\zeta,\xi,\rho,\mu} &= [\![\, t_1 \,]\!]_{\zeta,\xi,\rho,\mu}\ [\![\, t_2 \,]\!]_{\zeta,\xi,\rho,\mu} \\
[\![\, \lambda(x:\sigma).\ t \,]\!]_{\zeta,\xi,\rho,\mu} &= \lambda v \in [\![\sigma]\!]_{\zeta,\xi,\rho}.\ [\![\, t \,]\!]_{\zeta,\xi,\rho,\mu[x\mapsto v]} \\
[\![\, \lambda{:}(\alpha : k :\leq r).\ t \,]\!]_{\zeta,\xi,\rho,\mu} &= \lambda T \in [\![k]\!]_{\xi} \Downarrow [\![r]\!]_{\zeta}.\ [\![t]\!]_{\zeta,\xi,\rho[\alpha\mapsto T\uparrow[\![r]\!]_{\zeta}],\mu} \\
[\![\, t\,[:\sigma:] \,]\!]_{\zeta,\xi,\rho,\mu} &= [\![t]\!]_{\zeta,\xi,\rho,\mu}\ ([\![\sigma]\!]_{\zeta,\xi,\rho} \downarrow [\![r]\!]_{\zeta})
\end{aligned}$$

where for $t\,[:\sigma:]$, the type of t must have the form $\forall\alpha.\sigma'$, and r is the rank of α.

4 Primitive Rules of Inference of the HOL-Omega Logic

HOL-Omega includes all of the axioms and rules of inference of HOL, reinterpreting them in light of the expanded sets of types and terms, and extends them with the following new rules of inference, directed at the new varieties of terms.

– Rule `INST_TYPE` is revised; it says that consistently and properly substituting types for free type variables throughout a theorem yields a theorem.

$$\frac{\Gamma\ \vdash\ t}{\Gamma[\sigma_1,\ \ldots\ ,\sigma_n/\alpha_1,\ \ldots\ ,\alpha_n]\ \vdash\ t[\sigma_1,\ \ldots\ ,\sigma_n/\alpha_1,\ \ldots\ ,\alpha_n]} \qquad (\texttt{INST\_TYPE})$$

– Rule `INST_KIND` says that consistently substituting kinds for kind variables throughout a theorem yields a theorem.

$$\frac{\Gamma\ \vdash\ t}{\Gamma[k_1,\ \ldots\ ,k_n/\kappa_1,\ \ldots\ ,\kappa_n]\ \vdash\ t[k_1,\ \ldots\ ,k_n/\kappa_1,\ \ldots\ ,\kappa_n]} \qquad (\texttt{INST\_KIND})$$

– Rule `INST_RANK` says that consistently incrementing by $n \geq 0$ the rank of all type variables throughout a theorem yields a theorem. z is the rank variable.

$$\frac{\Gamma\ \vdash\ t}{\Gamma[(z+n)/z]\ \vdash\ t[(z+n)/z]} \qquad (\texttt{INST\_RANK})$$

– Rule `TY_ABS` says that if two terms are equal, then their type abstractions are equal, where α is not free in Γ.

$$\frac{\Gamma\ \vdash\ t_1 = t_2}{\Gamma\ \vdash\ (\lambda{:}\alpha.t_1) = (\lambda{:}\alpha.t_2)} \qquad (\texttt{TY\_ABS})$$

– Rule `TY_BETA_CONV` describes the equality of type beta-conversion, where $t[\sigma/\alpha]$ denotes the result of substituting σ for free occurrences of α in t.

$$\frac{}{\vdash\ (\lambda{:}\alpha.t)[:\sigma:]\ =\ t[\sigma/\alpha]} \qquad (\texttt{TY\_BETA\_CONV})$$

HOL-Omega adds one new axiom.

– Axiom `TY_ETA_AX` says type eta reduction is valid.

$$\vdash\ (\lambda{:}\alpha{:}\kappa.\ t[{:}\alpha{:}]) = t \qquad (\texttt{TY\_ETA\_AX})$$

To ensure the soundness of the HOL-Omega logic, all of the axioms and rules of inference need to have their semantic interpretations proven sound within set theory for all rank, kind, type, and term environments. This has not yet been formally done, but it is a priority for future work. When this is accomplished, by the LCF approach, all theorems proven within HOL-Omega will be sound.

5 Additional Type and Term Definitions

Of course the core of any system is only a point from which to begin. This section describes new type abbreviations and term constants not in HOL, defined as conservative extensions of the core logic of HOL-Omega.

5.1 New Type Abbreviations

HOL-Omega introduces the type abbreviations

$$\begin{array}{lcl} \mathsf{I} & = & \lambda(\alpha : 'k).\ \alpha \\ \mathsf{K} & = & \lambda(\alpha : 'k)\ (\beta : 'l).\ \alpha \\ \mathsf{S} & = & \lambda(\alpha : 'k \Rightarrow 'l \Rightarrow 'm)\ (\beta : 'k \Rightarrow 'l)\ (\gamma : 'k).\ \ \gamma\ \beta\ (\gamma\ \alpha) \\ \mathsf{o} & = & \lambda('f : 'k \Rightarrow 'l)\ ('g : 'l \Rightarrow 'm)\ (\alpha : 'k).\ \ \alpha\ 'f\ 'g \end{array}$$

The use of kind variables $'k$, $'l$, and $'m$ makes these type abbreviations applicable as type operators to types with arrow kinds. o is an infix type operator, written as $'f\ \mathsf{o}\ 'g = \lambda\alpha.\ \alpha\ 'f\ 'g$. These are reminiscent of the term combinators, e.g. $\texttt{I} = \lambda(x{:}\alpha).x$, $\texttt{K} = \lambda(x{:}\alpha)(y{:}\beta).x$, and $(g : \beta \to \gamma)\circ(f : \alpha \to \beta) = \lambda x.\ g\ (f\ x)$.

In HOL2P, both the arguments and the results of type operator applications must have the base kind **ty**. In HOL-Omega, the arguments and results may themselves be type operators of higher kind, as managed by the kind structure. This is of great advantage, for example when using o to compose two type operators, neither of which is applied to any arguments yet.

5.2 New Terms

HOL-Omega provides universal and existential quantification of type variables over terms using the new type binder constants $\forall{:}$ and $\exists{:}$, defined as

$$\begin{array}{lcl} \forall{:} & = & \lambda P.\ (P = (\lambda{:}\alpha{:}\kappa.\ \texttt{T})) \\ \exists{:} & = & \lambda P.\ (P \neq (\lambda{:}\alpha{:}\kappa.\ \texttt{F})) \end{array}$$

To ease readability, the following forms are also supported:

$$\begin{array}{lcl@{\qquad}lcl} \forall{:}\alpha{:}\kappa.\ P & = & \forall{:}\ (\lambda\alpha{:}\kappa.\ P) & \forall{:}\alpha_1{:}\kappa_1\ \alpha_2{:}\kappa_2\ \dots\ .\ P & = & \forall{:}\alpha_1{:}\kappa_1.\ \forall{:}\alpha_2{:}\kappa_2.\ \dots\ P \\ \exists{:}\alpha{:}\kappa.\ P & = & \exists{:}\ (\lambda\alpha{:}\kappa.\ P) & \exists{:}\alpha_1{:}\kappa_1\ \alpha_2{:}\kappa_2\ \dots\ .\ P & = & \exists{:}\alpha_1{:}\kappa_1.\ \exists{:}\alpha_2{:}\kappa_2.\ \dots\ P \end{array}$$

6 Examples

The HOL-Omega logic makes it straightforward to express many concepts from category theory, such as functors and natural transformations. Much of the first two examples below is ported from HOL2P [11]; the main difference is that the higher-order type abbreviations and type inference of HOL-Omega allow a more pleasing presentation. We focus on the category **Type** whose objects are the proper types of the HOL-Omega logic, and whose arrows are the (total) term functions from one type to another. The source and target of an arrow are the domain and range of the type of the function. The identity arrows are the identity functions on each type. The composition of arrows is normal functional composition. The customary check that the target of one arrow is the source of the other is accomplished automatically by the strong typing of the logic.

6.1 Functors

Functors map objects to objects and arrows to arrows. In the category **Type**, the first mapping is represented as a type $'F$ of kind $\mathbf{ty} \Rightarrow \mathbf{ty}$, and the second as a function of the type $'F$ $\mathsf{functor}$, where $\mathsf{functor}$ is the type abbreviation

$$\mathsf{functor} = \lambda' F.\ \forall \alpha\ \beta.\ (\alpha \to \beta) \to (\alpha\ 'F \to \beta\ 'F).$$

To be a functor, a function of this type must satisfy the following predicate:

$$\begin{array}{ll} \mathit{functor}\ (F : {'F}\ \mathsf{functor}) = & \\ \quad (\forall{:}\alpha.\ F\ (\mathtt{I} : \alpha \to \alpha) = \mathtt{I})\ \wedge & \mathit{Identity} \\ \quad (\forall{:}\alpha\ \beta\ \gamma.\ \forall (f : \alpha \to \beta)(g : \beta \to \gamma).\ F\ (g \circ f) = F\ g \circ F\ f\ ) & \mathit{Composition} \end{array}$$

where $g \circ f = \lambda x.\ g\ (f\ x)$. This is actually an abbreviated version; the parser and type inference fill in the necessary type applications, so the full version is

$$\begin{array}{ll} \mathit{functor}\ (F : {'F}\ \mathsf{functor}) = & \\ \quad (\forall{:}\alpha.\ F\,[{:}\alpha, \alpha{:}]\,(\mathtt{I} : \alpha \to \alpha) = \mathtt{I})\ \wedge & \mathit{Identity} \\ \quad (\forall{:}\alpha\ \beta\ \gamma.\ \forall (f : \alpha \to \beta)(g : \beta \to \gamma). & \mathit{Composition} \\ \qquad\qquad F\,[{:}\alpha, \gamma{:}]\,(g \circ f) = F\,[{:}\beta, \gamma{:}]\,g \circ F\,[{:}\alpha, \beta{:}]\,f\ ) & \end{array}$$

In what follows, these type applications will normally be omitted for clarity.

In HOL, $\mathtt{list} : \mathbf{ty} \Rightarrow \mathbf{ty}$ is the type of finite lists. It is defined as a recursive datatype with two constructors, $\mathtt{[]} : \alpha\ \mathtt{list}$ and $\mathtt{::} : \alpha \to \alpha\ \mathtt{list} \to \alpha\ \mathtt{list}$. $\mathtt{::}$ is infix. The function $\mathtt{MAP} : (\alpha \to \beta) \to (\alpha\ \mathtt{list} \to \beta\ \mathtt{list})$ is defined by

$$\begin{array}{lcl} \mathtt{MAP}\ f\ \mathtt{[]} & = & \mathtt{[]} \\ \mathtt{MAP}\ f\ (x\ \mathtt{::}\ xs) & = & f\ x\ \mathtt{::}\ \mathtt{MAP}\ f\ xs \end{array}$$

Then $\mathtt{MAP}$ can be proven to be a functor: $\vdash \mathit{functor}\ ((\lambda{:}\alpha\ \beta.\ \mathtt{MAP}) : \mathtt{list}\ \mathsf{functor})$.

A simple functor is the identity function $\mathtt{I}$: $\vdash \mathit{functor}\ ((\lambda{:}\alpha\ \beta.\ \mathtt{I}) : \mathsf{I}\ \mathsf{functor})$.

The composition of two functors is a functor. We overload $\circ$ to define this:

$$(G : {'G}\ \mathsf{functor}) \circ (F : {'F}\ \mathsf{functor}) = \lambda{:}\alpha\ \beta.\ G[{:}\alpha\ 'F, \beta\ 'F{:}] \circ F[{:}\alpha, \beta{:}]$$

The result has type $('F \text{ o } 'G)\mathsf{functor}$. As an example, $(\lambda{:}\alpha\,\beta.\ \mathtt{MAP}) \circ (\lambda{:}\alpha\,\beta.\ \mathtt{MAP}) = (\lambda{:}\alpha\ \beta.\ \mathtt{MAP} \circ \mathtt{MAP}) : (\mathtt{list}\text{ o }\mathtt{list})\mathsf{functor}$ is a functor. The type composition operator o reflects the category theory composition of two functors' mappings on objects. In HOL2P, the `MAP` functor composition example is expressed as:

$$\vdash \mathtt{TYINST}\ (\theta \mapsto \lambda\alpha.\ (\alpha\ \mathtt{list})\mathtt{list})\ \mathit{functor}\ (\lambda{:}\alpha\ \beta.\ \lambda f.\ \mathtt{MAP}\ (\mathtt{MAP}\ f))$$

Here the notation has been adjusted to that of this paper, for ease of comparison. `TYINST` is needed to manually instantiate a free type variable θ of the *functor* predicate with the type for this instance, which must be stated as a type abstraction. HOL-Omega's kinds and type inference enable a clearer statement:

$$\vdash \mathit{functor}\ (\lambda{:}\alpha\ \beta.\ \mathtt{MAP} \circ \mathtt{MAP})$$

Beyond the power of HOL2P, HOL-Omega supports quantification over functors:

$$\vdash \exists{:}'F.\ \exists(F : {'F}\ \mathsf{functor}).\ \mathit{functor}\ F.$$

6.2 Natural Transformations

Given functors F and G, a natural transformation maps objects A to arrows $FA \to GA$. In the category **Type**, we represent natural transformations as functions of the type $('F,'G)\mathsf{nattransf}$, where $\mathsf{nattransf}$ is the type abbreviation

$$\mathsf{nattransf} = \lambda'F\ 'G.\ \forall\alpha.\ \alpha\ 'F \to \alpha\ 'G.$$

A natural transformation ϕ from a functor F to a functor G ($\phi : F \to G$) must satisfy the following predicate:

$$\begin{aligned}&\mathit{nattransf}\,(\phi : ('F,'G)\mathsf{nattransf})\ (F : {'F}\ \mathsf{functor})\ (G : {'G}\ \mathsf{functor}) = \\ &\qquad \forall{:}\alpha\ \beta.\ \forall(h : \alpha \to \beta).\ \ G\ h \circ \phi = \phi \circ F\ h\end{aligned}$$

Define the function `INITS` to take a list and return a list of all prefixes of it:

$$\begin{aligned}\mathtt{INITS}\ \mathtt{[]} &= \mathtt{[]}\\ \mathtt{INITS}\ (x\ \mathtt{::}\ xs) &= \mathtt{[]}\ \mathtt{::}\ \mathtt{MAP}\ (\lambda ys.\ x\ \mathtt{::}\ ys)\ (\mathtt{INITS}\ xs)\end{aligned}$$

`INITS` can be proven to be a natural transformation from `MAP` to `MAP` $\circ$ `MAP`:

$$\begin{aligned}\vdash \mathit{nattransf}\ &((\lambda{:}\alpha.\quad \mathtt{INITS}) &&: (\mathtt{list},\ \mathtt{list}\text{ o }\mathtt{list})\mathsf{nattransf})\\ &((\lambda{:}\alpha\ \beta.\ \ \mathtt{MAP}) &&:\ \mathtt{list}\ \mathsf{functor})\\ &((\lambda{:}\alpha\ \beta.\ \ \mathtt{MAP} \circ \mathtt{MAP}) &&: (\mathtt{list}\text{ o }\mathtt{list})\mathsf{functor}).\end{aligned}$$

The vertical composition of two natural transformations is defined as

$$(\phi_2 : ('G,'H)\mathsf{nattransf}) \circ (\phi_1 : ('F,'G)\mathsf{nattransf}) \ = \ \lambda{:}\alpha.\ \phi_2 \circ (\phi_1[{:}\alpha{:}])$$

The result of this vertical composition is a natural transformation:

$$\begin{aligned}\vdash \mathit{nattransf}\,(&\quad \phi_1 &&: ('F,'G)\mathsf{nattransf})\ \ F\ \ G\ \ \wedge\\ \mathit{nattransf}\,(&\quad \phi_2 &&: ('G,'H)\mathsf{nattransf})\ \ G\ \ H\ \ \Rightarrow\\ \mathit{nattransf}\,(&\ \phi_2 \circ \phi_1 &&: ('F,'H)\mathsf{nattransf})\ \ F\ \ H\end{aligned}$$

A natural transformation may be composed with a functor in two ways, where the functor is either applied first or last. We define these, again overloading $\circ$:

$$\begin{array}{lll}(\phi : ('F,'G)\mathsf{nattransf}) \circ (H : 'H\ \mathsf{functor}) & = & \lambda{:}\alpha.\ \phi\ [{:}\alpha\ 'H{:}] \\ (H : 'H\ \mathsf{functor}) \circ (\phi : ('F,'G)\mathsf{nattransf}) & = & \lambda{:}\alpha.\ H\ (\phi\ [{:}\alpha{:}])\end{array}$$

That the last of these is a natural transformation is expressed in HOL2P as

$$\begin{array}{l}\vdash \mathit{nattransf}\ \phi\ F\ G\ \wedge\ \mathit{functor}\ H\ \Rightarrow \\ \quad \mathtt{TYINST}\ ((\theta_1 \mapsto \lambda\alpha.\ ((\alpha)\theta_1)\theta_3)\ (\theta_2 \mapsto \lambda\alpha.\ ((\alpha)\theta_2)\theta_3)) \\ \qquad \mathit{nattransf}\,(\lambda{:}\alpha.\ H\ \phi)\ (\lambda{:}\alpha\ \beta.\ H \circ F)\ (\lambda{:}\alpha\ \beta.\ H \circ G)\end{array}$$

where in HOL-Omega, the type inference, higher kinds, and overloaded $\circ$ permit

$$\begin{array}{l}\vdash \mathit{nattransf}\ \phi\ F\ G\ \wedge\ \mathit{functor}\ H\ \Rightarrow \\ \quad \mathit{nattransf}\,(H \circ \phi)\ (H \circ F)\ (H \circ G).\end{array}$$

6.3 Monads

Wadler [12] has proposed using monads to structure functional programming. He defines a monad as a triple $('M,\ \mathit{unit},\ \gg\!=)$ of a type operator $'M$ and two term operators *unit* and $\gg\!=$ (where $\gg\!=$ is an infix operator) obeying three laws. We express this definition in HOL-Omega as follows.

We define two type abbreviations unit and bind:

$$\begin{array}{rl}\mathsf{unit} = & \lambda'M.\ \forall\alpha.\ \alpha\ \to \alpha\ 'M \\ \mathsf{bind} = & \lambda'M.\ \forall\alpha\ \beta.\ \alpha\ 'M \to (\alpha \to \beta\ 'M) \to \beta\ 'M\end{array}$$

We define a monad to be two term operators, *unit* and $\gg\!=$, with a single common free type variable $'M : \mathbf{ty}{\Rightarrow}\mathbf{ty}$, satisfying a predicate of the three laws:

$$\begin{array}{llr}\multicolumn{3}{l}{\mathit{monad}\ (\mathit{unit} : 'M\ \mathsf{unit},\ \gg\!= : 'M\ \mathsf{bind}) =} \\ \quad (\forall{:}\alpha\ \beta. & \forall(a : \alpha)(k : \alpha \to \beta\ 'M). & \textit{(Left unit)} \\ & \quad \mathit{unit}\ a \gg\!= k\ =\ k\ a)\ \wedge & \\ \quad (\forall{:}\alpha. & \forall(m : \alpha\ 'M). & \textit{(Right unit)} \\ & \quad m \gg\!= \mathit{unit}\ =\ m)\ \wedge & \\ \quad (\forall{:}\alpha\ \beta\ \gamma. & \forall(m : \alpha\ 'M)(k : \alpha \to \beta\ 'M)(h : \beta \to \gamma\ 'M). & \textit{(Associative)} \\ & \quad (m \gg\!= k) \gg\!= h\ =\ m \gg\!= (\lambda\alpha.\ k\ \alpha \gg\!= h)) & \end{array}$$

As an example, we define the *unit* and $\gg\!=$ operations for a state monad as

$$\mathsf{state} = \lambda\sigma\ \alpha.\ \sigma \to \alpha \times \sigma$$

$$\begin{array}{l}\mathit{state\_unit} = \lambda{:}\alpha.\ \lambda(x{:}\alpha)\ (s{:}\sigma).\ (x, s) \\ \mathit{state\_bind} = \lambda{:}\alpha\ \beta.\ \lambda(w{:}(\sigma,\alpha)\mathsf{state})\ (f{:}\alpha{\to}(\sigma,\beta)\mathsf{state})\ (s{:}\sigma).\ \mathbf{let}\ (x, s') = w\ s \\ \hspace{18em} \mathbf{in}\ f\ x\ s'\end{array}$$

Then we can prove these operations satisfy the *monad* predicate for $'M = \sigma\ \mathsf{state}$, taking advantage of the curried nature of state, where $(\sigma,\alpha)\mathsf{state} = \alpha\ (\sigma\ \mathsf{state})$:

$$\vdash \mathit{monad}\ (\ \mathit{state\_unit} : (\sigma\ \mathsf{state})\mathsf{unit},\ \ \mathit{state\_bind} : (\sigma\ \mathsf{state})\mathsf{bind}\ ).$$

Wadler [12] also formulates an alternative definition of monads, expressed in terms of three operators, *unit*, *map*, and *join*, satisfying seven laws:

$$\begin{array}{l} \mathsf{map} = \lambda' M.\ \forall \alpha\ \beta.\ (\alpha \to \beta) \to (\alpha\ 'M \to \beta\ 'M) \\ \mathsf{join} = \lambda' M.\ \forall \alpha.\ \alpha\ 'M\ 'M \to \alpha\ 'M \end{array}$$

$$\begin{array}{llll}
\multicolumn{4}{l}{\mathit{umj\_monad}\ (\mathit{unit} : 'M\ \mathsf{unit},\ \mathit{map} : 'M\ \mathsf{map},\ \mathit{join} : 'M\ \mathsf{join}) =} \\
\quad (\forall{:}\alpha. & \mathit{map}\ (\mathtt{I} : \alpha \to \alpha) = \mathtt{I}) & \wedge & \mathit{(map\_I)} \\
\quad (\forall{:}\alpha\ \beta\ \gamma.\ \forall (f : \alpha \to \beta)(g : \beta \to \gamma). & & & \mathit{(map\_o)} \\
 & \mathit{map}\ (g \circ f) = \mathit{map}\ g \circ \mathit{map}\ f) & \wedge & \\
\quad (\forall{:}\alpha\ \beta.\quad \forall (f : \alpha \to \beta). & & & \mathit{(map\_unit)} \\
 & \mathit{map}\ f \circ \mathit{unit} = \mathit{unit} \circ f) & \wedge & \\
\quad (\forall{:}\alpha\ \beta.\quad \forall (f : \alpha \to \beta). & & & \mathit{(map\_join)} \\
 & \mathit{map}\ f \circ \mathit{join} = \mathit{join} \circ \mathit{map}\ (\mathit{map}\ f)) & \wedge & \\
\quad (\forall{:}\alpha. & \mathit{join} \circ \mathit{unit} = (\mathtt{I} : \alpha\ 'M \to \alpha\ 'M)) & \wedge & \mathit{(join\_unit)} \\
\quad (\forall{:}\alpha. & \mathit{join} \circ \mathit{map}\ \mathit{unit} = (\mathtt{I} : \alpha\ 'M \to \alpha\ 'M)) & \wedge & \mathit{(join\_map\_unit)} \\
\quad (\forall{:}\alpha. & \mathit{join}\ [{:}\alpha{:}] \circ \mathit{map}\ \mathit{join} = \mathit{join} \circ \mathit{join}) & & \mathit{(join\_map\_join)}
\end{array}$$

Given a monad defined using *unit* and $\gg\!=$, corresponding *map* and *join* operators $\mathtt{MMAP}(\mathit{unit}, \gg\!=)$ and $\mathtt{JOIN}(\mathit{unit}, \gg\!=)$ may be constructed automatically:

$$\begin{array}{l}
\mathtt{MMAP}\ (\mathit{unit} : 'M\ \mathsf{unit},\ \gg\!= : 'M\ \mathsf{bind}) \\
\qquad =\ \lambda{:}\alpha\ \beta.\ \lambda (f : \alpha \to \beta)\ (m : \alpha\ 'M).\ m \gg\!= (\lambda a.\ \mathit{unit}\ (f\ a)) \\
\mathtt{JOIN}\ (\mathit{unit} : 'M\ \mathsf{unit},\ \gg\!= : 'M\ \mathsf{bind}) \\
\qquad =\ \lambda{:}\alpha.\ \lambda (z : \alpha\ 'M\ 'M).\ z \gg\!= \mathtt{I}
\end{array}$$

Given a monad defined using *unit*, *map*, and *join*, the corresponding $\gg\!=$ operator $\mathtt{BIND}(\mathit{map}, \mathit{join})$ may also be constructed automatically:

$$\begin{array}{l}
\mathtt{BIND}\ (\mathit{map} : 'M\ \mathsf{map},\ \mathit{join} : 'M\ \mathsf{join}) \\
\qquad =\ \lambda{:}\alpha\ \beta.\ \lambda (m : \alpha\ 'M)\ (k : \alpha \to \beta\ 'M).\ \mathit{join}\ (\mathit{map}\ k\ m)
\end{array}$$

E.g., for the state monad, $\mathit{state\_map} = \mathtt{MMAP}\ (\mathit{state\_unit},\ \mathit{state\_bind})$
$\mathit{state\_join} = \mathtt{JOIN}\ (\mathit{state\_unit},\ \mathit{state\_bind})$
$\mathit{state\_bind} = \mathtt{BIND}\ (\mathit{state\_map},\ \mathit{state\_join})$.

Then it can be proven that these two definitions of a monad are equivalent.

$$\begin{array}{l}
\vdash\ \mathit{monad}\ (\mathit{unit} : 'M\ \mathsf{unit},\ \gg\!= : 'M\ \mathsf{bind})\ \Leftrightarrow \\
\quad (\mathit{umj\_monad}\ (\mathit{unit},\ \mathtt{MMAP}(\mathit{unit}, \gg\!=),\ \mathtt{JOIN}(\mathit{unit}, \gg\!=))\ \wedge \\
\qquad \gg\!=\ =\ \mathtt{BIND}\ (\mathtt{MMAP}(\mathit{unit}, \gg\!=),\ \mathtt{JOIN}(\mathit{unit}, \gg\!=)))
\end{array}$$

$$\begin{array}{l}
\vdash\ \mathit{umj\_monad}(\mathit{unit} : 'M\ \mathsf{unit},\ \mathit{map} : 'M\ \mathsf{map},\ \mathit{join} : 'M\ \mathsf{join})\ \Rightarrow \\
\quad \mathit{monad}(\mathit{unit},\ \mathtt{BIND}(\mathit{map}, \mathit{join}))
\end{array}$$

Lack and Street [7] define monads as a category A, a functor $t : A \to A$, and natural transformations $\mu : t^2 \to t$ and $\eta : 1_A \to t$ satisfying three equations, as expressed by the commutative diagrams (in the functor category)

$$\begin{tikzcd}
t^3 \arrow[r, "t\mu"] \arrow[d, "\mu t"'] & t^2 \arrow[d, "\mu"] \\
t^2 \arrow[r, "\mu"'] & t
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
t \arrow[r, "t\eta"] \arrow[dr, "1"'] & t^2 \arrow[d, "\mu"] & t \arrow[l, "\eta t"'] \arrow[dl, "1"] \\
& t. &
\end{tikzcd}$$

This definition can be expressed in HOL-Omega as follows:

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathit{cat\_monad}\ (t : {'M}\ \mathsf{functor},\ \mu : ({'M}\ \mathsf{o}\ {'M},\ {'M})\mathsf{nattransf},\ \eta : (\mathsf{I},\ {'M})\mathsf{nattransf}) =} \\
\quad \mathit{functor}\ t & \wedge & \textit{(t is a functor)} \\
\quad \mathit{nattransf}\ \mu\ (t \circ t)\ t & \wedge & \textit{(μ is a natural transformation)} \\
\quad \mathit{nattransf}\ \eta\ (\lambda{:}\alpha\ \beta.\ \mathtt{I})\ t & \wedge & \textit{(η is a natural transformation)} \\
\quad (\mu \circ (t \circ \mu) = \mu \circ (\mu \circ t)) & \wedge & \textit{(square commutes)} \\
\quad (\mu \circ (t \circ \eta) = \lambda{:}\alpha.\ \mathtt{I}) & \wedge & \textit{(left triangle commutes)} \\
\quad (\mu \circ (\eta \circ t) = \lambda{:}\alpha.\ \mathtt{I}) & & \textit{(right triangle commutes).}
\end{array}$$

It can be proven that this is equivalent to the (*unit*, *map*, *join*) definition:

$$\begin{array}{l}
\vdash \forall(\mathit{unit} : {'M}\ \mathsf{unit})\ \mathit{map}\ \mathit{join}. \\
\quad \mathit{cat\_monad}(\mathit{map},\ \mathit{join},\ \mathit{unit}) \iff \mathit{umj\_monad}(\mathit{unit},\ \mathit{map},\ \mathit{join}).
\end{array}$$

Therefore all three definitions of monads are equivalent.

7 Conclusion

This document has presented a description of the core logic of the HOL-Omega theorem prover. This has been implemented as a variant of the HOL4 theorem prover. The implementation may be downloaded by the command

```
svn checkout https://hol.svn.sf.net/svnroot/hol/branches/HOL-Omega
```

Installation instructions are in the top directory.

This provides a practical workbench for developments in the HOL-Omega logic, integrated in a natural and consistent manner with the existing HOL4 tools and libraries that have been refined and extended over many years.

This implementation was designed with particular concern for backward compatibility.This was almost entirely achieved, which was possible only because the fundamental data types representing types and terms were originally encapsulated. This meant that the underlying representation could be changed without affecting the abstract view of types and terms by the rest of the system. Virtually all existing HOL4 code will build correctly, including the extensive libraries. The simplifiers have been upgraded, including higher-order matching of the new types and terms and automatic type beta-reduction. Algebraic types with higher kinds and ranks may be constructed using the familiar `Hol_datatype` tool [5]. Not all of the tools will work as expected on the new terms and types, as the revision process is ongoing, but they will function identically on the classic terms and types. So nothing of HOL4's power has been lost.

Also, the nimble ease of use of HOL has been largely preserved. For example, the type inference algorithm is a pure extension, so that all classic terms have the same types successfully inferred. Inference of most general types for all terms is not always possible, as also seen in System F, and type inference may fail even for typeable terms, but in practice a few user annotations are usually sufficient.

The system is still being developed but is currently useful. All of the examples presented have been mechanized in the `examples/HolOmega` subdirectory, along with further examples from *Algebra of Programming* [1] ported straightforwardly from HOL2P, including homomorphisms, initial algebras, catamorphisms, and the banana split theorem. While maintaining backwards compatibility with the existing HOL4 system and libraries, the additional expressivity and power of HOL-Omega makes this tool applicable to a great collection of new problems.

Acknowledgements. Norbert Völker's HOL2P [11] was an vital inspiration. Michael Norrish helped get the new branch of HOL4 established and to begin the new parsers and prettyprinters. John Matthews suggested adding the single rank variable to every rank. William Schneeburger justified an aggressive set-theoretic semantics of ranks. Mike Gordon has consistently encouraged this work. We honor his groundbreaking and seminal achievement in the original HOL system [3], without which none of this work would have been possible.

Soli Deo Gloria.

References

1. Bird, R., de Moor, O.: Algebra of Programming. Prentice Hall (1997)
2. Coquand, T.: A new paradox in type theory. In: Prawitx, D., Skyrms, B., Westerstahl, D. (eds.) Proceedings 9th Int. Congress of Logic, Methodology and Philosophy of Science, pp. 555–570. North-Holland, Amsterdam (1994)
3. Gordon, M.J.C., Melham, T.F.: Introduction to HOL. Cambridge University Press, Cambridge (1993)
4. Gordon, M.J.C., Pitts, A.M.: The HOL Logic and System. In: Bowen, J. (ed.) Towards Verified Systems, ch. 3, pp. 49–70. Elsevier Science B.V., Amsterdam (1994)
5. The HOL System DESCRIPTION (Version Kananaskis 4), `http://downloads.sourceforge.net/hol/kananaskis-4-description.pdf`
6. The HOL System LOGIC (Version Kananaskis 4), `http://downloads.sourceforge.net/hol/kananaskis-4-logic.pdf`
7. Lack, S., Street, R.: The formal theory of monads II. Journal of Pure Applied Algorithms 175, 243–265 (2002)
8. Melham, T.F.: The HOL Logic Extended with Quantification over Type Variables. Formal Methods in System Design 3(1-2), 7–24 (1993)
9. Monk, J.D.: Introduction to Set Theory. McGraw-Hill, New York (1969)
10. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge (2002)
11. Völker, N.: HOL2P - A System of Classical Higher Order Logic with Second Order Polymorphism. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 334–351. Springer, Heidelberg (2007)
12. Wadler, P.: Monads for functional programming. In: Jeuring, J., Meijer, E. (eds.) AFP 1995. LNCS, vol. 925. Springer, Heidelberg (1995)