The HOL-Omega Logic

August, 2011

Dr. Peter V. Homeier

palantir@trustworthytools.com

U.S. Department of Defense

Making formal methods
into normal methods

Outline

- Motivation
- Higher order logic (HOL) (*including the lambda calculus*)
  - Types: Type constants, variables
  - Terms: constants, variables, applications, abstractions
- Type operators
  - Kinds
  - Types: adds abstractions of types, which have higher kinds
- System F
  - Types: adds universal types
  - Terms: adds type abstractions, type applications
- HOL-Omega logic = HOL + Type operators + System F
- Applications: category theory and monads

Motivation: Monads

- Monads are a practical device to cleanly represent computations that involve state, I/O, exceptions, …
- A rich theory exists with many different kinds of monads
- Individual monads can (and have) been defined in HOL
- A general theory of monads ***cannot*** be defined in HOL
- Why? A *monad* is a type operator *M* together with two term operations *unit* and >>= (*"bind"*, infix) such that

  1) $unit\ a >>= k = k\ a$ *left unit*

  2) $m >>= unit = m$ *right unit*

  3) $(m >>= k) >>= h = m >>= (\lambda a.\ k\ a >>= h)$ *associativity*

- In law (3), >>= has 4 occurrences and 3 distinct types, but HOL requires a variable to have a single unique type!

Higher Order Logic (HOL4)

(Church's Simple Theory of Types)

- Easy to use: natural to understand and simple to write:
  - Completely decidable type inference (Hindley-Milner)
  - Classical logic with quantifiers over h.o. functions, excluded middle, and choice
  - Function / predicate extensionality: $(\forall x.\ P\ x = Q\ x) \Rightarrow P = Q$
- Practical: used for many major projects
  - Faithful model of ARM microprocessor (Anthony Fox)
  - Large industrial-scale projects
- Powerful: many deep libraries and features:
  - Full total recursive function definition
  - Finite machine words library (e.g., bitvectors) using pseudo dependent types
  - Simplifier and two first order automatic theorem provers built-in
  - Higher order quotients with automatic lifting of types, constants, theorems
- Safe: LCF architecture, very small kernel, for high trustworthiness
- Mature: 23 years of development and application, still being actively developed
- Is imitation the sincerest form of flattery?
  - Isabelle/HOL, HOL-Light, ProofPower, HOL Zero all use essentially the same logic

Higher Order Logic (as in Gordon's HOL system, 1988)

- *Syntax*

  t ::= terms:

$c : \sigma$	constant
$x : \sigma$	variable
$\lambda x.\ t$	abstraction
$t_{opr}\ t_{arg}$	application

  σ ::= types:

α	type variable
$(\sigma_1,\ldots\ \sigma_n)\tau$	type combination

  θ ::= type substitution:

$[\,]$	empty substitution
$(\alpha \mid\!\rightarrow \sigma) :: \theta$	subst mapping

- *Environment* Γ

  stores type arities, constant types

  Types: bool, ind, fun ($\rightarrow$)

  Terms: $= : \alpha \rightarrow \alpha \rightarrow bool$

  $\Rightarrow : bool \rightarrow bool \rightarrow bool$

  $@ : (\alpha \rightarrow bool) \rightarrow \alpha$

- *Constraints*

  Types $(\sigma_1,\ldots\ \sigma_n)\tau$:
  arity of τ must match $n \geq 0$

  Application terms $t_{opr}\ t_{arg}$:
  type of t_{opr} must be a function type,
  and domain of type of t_{opr}
  must match type of t_{arg}

- *Typing*

  $\Gamma \vdash t : \sigma$

$$\frac{c{:}\sigma \in \Gamma}{\Gamma \vdash c : \sigma\,\theta}$$

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

$$\frac{\Gamma, x{:}\sigma_1 \vdash t_2 : \sigma_2}{\Gamma \vdash \lambda x : \sigma_1.\ t_2 : \sigma_1 \rightarrow \sigma_2}$$

$$\frac{\Gamma \vdash t_1 : \sigma_{11} \rightarrow \sigma_{12} \quad \Gamma \vdash t_2 : \sigma_{11}}{\Gamma \vdash t_2 : \sigma_{11}}$$

Lambda-Omega = HOL + type operators and kinds

- *Syntax*

t ::= terms:

$c : \sigma$	constant
$x : \sigma$	variable
$\lambda x.\ t$	abstraction
$t_{opr}\ t_{arg}$	application

σ ::= types:

$\alpha : k$	type variable
deleted $(\sigma_1, \ldots \sigma_n)\tau$	type
$\tau : k$	type constant
$\lambda\alpha.\ \sigma$	type abstraction
$\sigma_{arg}\ \sigma_{opr}$	type application

k ::= kinds:

ty	kind of proper types
$k_1 \Rightarrow k_2$	kind of type operators

θ ::= type substitution:

[ ]	empty substitution
$(\alpha \mid\rightarrow \sigma) :: \theta$	subst mapping

- *Constraints*

Application types $\sigma_{arg}\ \sigma_{opr}$:
kind of σ_{opr} must be an operator kind, and
domain of kind of σ_{opr} must match kind of σ_{arg}

Application terms $t_{opr}\ t_{arg}$:
type of t_{opr} must be a function type, and
domain of type of t_{opr} must match type of t_{arg}

- *Typing*

$\Gamma \mid\!- t : \sigma$

$$\frac{c{:}\sigma \in \Gamma}{\Gamma \mid\!- c : \sigma\ \theta}$$

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \mid\!- x : \sigma}$$

$$\frac{\Gamma \mid\!- \sigma_1 : \mathbf{ty} \qquad \Gamma, x{:}\ \sigma_1 \mid\!- t_2 : \sigma_2}{\Gamma \mid\!- \lambda x : \sigma_1.\ t_2\ : \sigma_1 \rightarrow \sigma_2}$$

$$\frac{\Gamma \mid\!- t_{opr} : \sigma_1 \rightarrow \sigma_2 \qquad \Gamma \mid\!- t_{arg} : \sigma_1}{\Gamma \mid\!- t_{opr}\ t_{arg} : \sigma_2}$$

$$\frac{\Gamma \mid\!- t : \sigma_1 \qquad \sigma_1 \equiv_{\alpha\beta} \sigma_2 \qquad \Gamma \mid\!- \sigma_2 : \mathbf{ty}}{\Gamma \mid\!- t : \sigma_2}$$

System F *(Girard, Reynolds)* adds type abstractions, applications

- *Syntax*

  t ::= terms:

$c : \sigma$	constant
$x : \sigma$	variable
$\lambda x.\ t$	abstraction
$t_{opr}\ t_{arg}$	application
$\lambda{:}\alpha.\ t$	type abstraction term
$t\ [: \sigma :]$	type application term

  σ ::= types:

α	type variable
$(\sigma_1,\ldots\ \sigma_n)\tau$	type combination
$\forall\alpha.\ \sigma$	universal type

  Impredicativity!

- *Constraints*

  Types $(\sigma_1,\ldots\ \sigma_n)\tau$:

  arity of τ must match $n \geq 0$

  Application terms $t_{opr}\ t_{arg}$:

  type of t_{opr} must be a function type, and

  domain of type of t_{opr} must match type of t_{arg}

- *Constraints (continued)*

  Type application terms $t\ [: \sigma :]$:

  type of t must be a universal type

- *Typing*

  $\Gamma \mid\!- t : \sigma$

$$\frac{c{:}\sigma \in \Gamma}{\Gamma \mid\!- c : \sigma}$$

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \mid\!- x : \sigma}$$

$$\frac{\Gamma, x{:}\ \sigma_1 \mid\!- t_2 : \sigma_2}{\Gamma \mid\!- \lambda x : \sigma_1.\ t_2 : \sigma_1 \rightarrow \sigma_2}$$

$$\frac{\Gamma \mid\!- t_{opr} : \sigma_1 \rightarrow \sigma_2 \quad \Gamma \mid\!- t_{arg} : \sigma_1}{\Gamma \mid\!- t_{opr}\ t_{arg} : \sigma_2}$$

$$\frac{\Gamma, \alpha \mid\!- t : \sigma}{\Gamma \mid\!- \lambda{:}\alpha.\ t : \forall\alpha.\ \sigma}$$

$$\frac{\Gamma \mid\!- t_1 : \forall\alpha.\ \sigma_1}{\Gamma \mid\!- t_1\ [: \sigma_2 :] : \sigma_1[\alpha \mid\!\rightarrow \sigma_2]}$$

HOL-Omega = HOL4 + System F, type operators, kinds, & ranks

Using ideas from HOL2P by Völker and F_ω by Pierce

- *Syntax*

t ::=		terms:
	$c : \sigma$	constant
	$x : \sigma$	variable
	$\lambda x.\ t$	abstraction
	$t_{opr}\ t_{arg}$	application
	$\lambda{:}\alpha.\ t$	type abstraction term
	$t\ [: \sigma :]$	type application term
σ ::=		types:
	$\alpha : k$	type variable
	$\tau : k$	type constant
	$\sigma_{arg}\ \sigma_{opr}$	type application
	$\lambda\alpha.\ \sigma$	type abstraction
	$\forall\alpha.\ \sigma$	universal type
	$\exists\alpha.\ \sigma$	existential type
k ::=		kinds:
	ty : r	kind of proper types
	κ : r	kind variable
	$k_1 \Rightarrow k_2$	kind of type operators
r ::= *n*		ranks: (natural nums)

- *Constraints*

Type application terms t [: σ :] :
type of t must be universal type, say $\forall\alpha.\ \sigma_1$
and kind of α must match kind of σ
and rank of α must be ≥ rank of σ

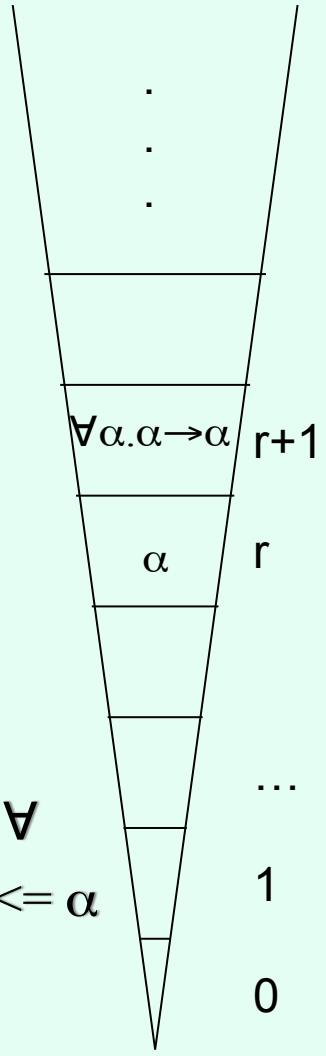
Application types $\sigma_{arg}\ \sigma_{opr}$:
kind of σ_{opr} must be an operator kind(⇒), and
domain of kind of σ_{opr} must match kind of σ_{arg}
and rank(domain(σ_{opr})) must ≥ rank(σ_{arg}).

Application terms $t_{opr}\ t_{arg}$:
type of t_{opr} must be a function type, and
domain of type of t_{opr} must match type of t_{arg}
and rank(domain(t_{opr})) must ≥ rank(t_{arg}).

- Ranks are present to avoid impredicativity, and so assure a set-theoretic model exists.
  Types are ***stratified*** by increasing rank:
  rank(α : k) = rank(k)
  rank(τ : k) = rank(k)
  rank($\sigma_{arg}\ \sigma_{opr}$) = rank(range($\sigma_{opr}$))
  rank(λα. σ) = max(rank(α), rank(σ))
  rank(∀α. σ) = max(rank(α) + 1, rank(σ))
  rank(∃α. σ) = max(rank(α) + 1, rank(σ))

Impredicativity

- A powerful form of parametric polymorphism, but dangerous. Similar in power to ZF set theory.
- A definition is "impredicative" if it involves ranging over a domain which includes the very thing being defined.
- In System F, the type variable α in the type $\sigma = \forall\alpha.\alpha\rightarrow\alpha$ ranges over all types, including σ itself.
- This is circular, but System F is *not* inconsistent.
- However, Girard discovered that the naïve combination of impredicativity and higher order logic is inconsistent!
- Our design choice: disallow impredicativity of types.
  - Stratify types by ranks 0, 1, 2, … according to depth of $\forall$
  - Type variable α in $\forall\alpha.\alpha\rightarrow\alpha$ ranges over types of rank <= α
  - Rank of the type $\forall\alpha.\alpha\rightarrow\alpha$ is then (rank of α) + 1
  - Yields straightforward set-theoretic semantics



Universal and Existential Quantification of type variables over terms

- HOL-Omega has abstraction of a type variable α over a term t
  - $\lambda{:}(\alpha{:}\kappa).\ t$ (type abstraction)
    - Note that the type variable can have any kind κ
- Type quantification over terms is defined using type abstraction:
  - $\forall{:} = \lambda P.\ (P = \lambda{:}(\alpha{:}\kappa).\ \mathtt{T})$
  - $\exists{:} = \lambda P.\ (P \neq \lambda{:}(\alpha{:}\kappa).\ \mathtt{F})$
- Notation:
  - $\forall{:}(\alpha{:}\kappa).\ t = (\forall{:})\ (\lambda{:}(\alpha{:}\kappa).\ t)$
  - $\exists{:}(\alpha{:}\kappa).\ t = (\exists{:})\ (\lambda{:}(\alpha{:}\kappa).\ t)$
- Multiple type abstraction, quantification, application:
  - $(\lambda{:}\alpha\ \beta.\ \lambda(x{:}\beta \rightarrow \alpha).\ x)\ [:\text{ bool, int }:] = \lambda(x{:}\ \text{int} \rightarrow \text{bool}).\ x$

Application: functors
(ported from HOL2P by Völker)

- Category **Type**: objects are types of kind **ty** & rank 0, arrows are term functions
- Functors: Two maps: 1) on objects *'F* : **ty⇒ty** and 2) on arrows *F* : *'F* functor
- Functor type abbreviation:
  functor = λ*'F* : **ty⇒ty**. ∀α β. (α → β) → (α *'F* → β *'F*)
- Functor predicate: true if *F* (of type *'F* functor) is a functor
  functor (*F* : *'F* functor) =
  1) (∀:α. *F* I = I) ∧ *(where* I *is the identity function* λ*x. x)*
  2) (∀:α β γ. ∀(*f*: β → γ) (*g*: α → β).
     F (*f* ° *g*) = *F f* ° *F g*)
- Full, unabbreviated version:
  functor (*F* : *'F* functor) =
  1) (∀:α. *F* [: α, α :] (I : α → α) = (I : α *'F* → α *'F*)) ∧
  2) (∀:α β γ. ∀(*f*: β → γ) (*g*: α → β).
     F [: α, γ :] (*f* ° *g*) = *F* [: β, γ :] *f* ° *F* [: α, β :] *g*)
- Provable in HOL-Omega (but not expressable in HOL2P):
  |– ∃:*'F*. ∃*F* : *'F* functor. *functor F*

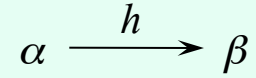
Theorems about functors

- Examples of functors:
  Identity functor *id* = λ:α β. (I : (α → β) → (α → β)) :
  ⊢ *functor* (*id* : I functor) *(where* I = λα:κ. α*)*
  `MAP` functor:
  ⊢ *functor* ((λ:α β. (`MAP` : (α → β) → (α list → β list))) : list functor)
  where `MAP` *f* [ ] = [ ]
  `MAP` *f* (*x* :: *xs*) = *f x* :: `MAP` *f xs*
- Define the composition of two functors (overloading °):
  (*G* :'*G* functor) ° (*F* :'*F* functor) = λ:α β. *G* [: α 'F, β 'F :] ° *F* [: α, β :]
- Proved in HOL-Omega that the composition of functors is also a functor:
  ⊢ *functor* (*F* :'*F* functor) ∧ *functor* (*G* :'*G* functor)
  ⇒ *functor* ((*G* ° *F*) : ('*F* o '*G*)functor)
  where the infix type operator o = λ('*F*:'*k*⇒'*l*)('*G*:'*l*⇒'*m*)(α:'*k*). (α '*F*)'*G*
- Example: the composition of the `MAP` functor with itself is a functor:
  ⊢ *functor* (((λ:α β. `MAP`) ° (λ:α β. `MAP`)) : (list o list)functor)
  (by defn.,) (λ:α β. `MAP`) ° (λ:α β. `MAP`) = (λ:α β. `MAP` ° `MAP`)

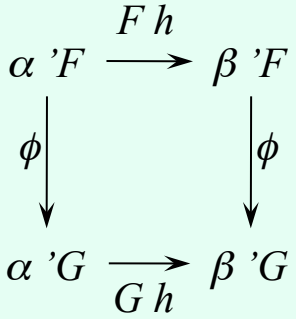
Application: natural transformations

(from Völker's HOL2P and Algebra of Programming)

- Natural transformation type abbreviation:
  nattransf = λ(*'F* :**ty⇒ty**) (*'G* :**ty⇒ty**). ∀α. α *'F* → α *'G*



- Natural transformation predicate:
  nattransf (φ : (*'F*, *'G*)nattransf) (*F* : *'F* functor) (*G* : *'G* functor) =
  ∀:α β. ∀(*h*: α → β). *G h* ° φ = φ ° *F h*



- Read as "φ is a natural transformation from functor *F* to functor *G*"
- Identity natural transformation from any functor to itself:
  id_F = (λ:α. I : α *'F* → α *'F*) : (*'F*, *'F*)nattransf
  |– *nattransf* (id_F :(*'F*, *'F*)nattransf) (*F* : *'F* functor) *F*
- Example: `INITS` returns a list of all prefixes of its argument:

  `INITS` : α list → α list list

  `INITS` [ ] = [ ]

  `INITS` (*x* :: *xs*) = [ ] :: `MAP` (λ*ys*. *x* :: *ys*) (`INITS` *xs*)
- `INITS` is a natural transformation from `MAP` to `MAP` ° `MAP`:
  |– *nattransf* ((λ:α. `INITS`) : (list, list o list)nattransf)
  ((λ:α β. `MAP`) : list functor)
  ((λ:α β. `MAP` ° `MAP`) : (list o list)functor)

Composing natural transformations

- Vertical composition of natural transformations:
  $(\phi_2 : ('G,'H)\text{nattransf}) \circ (\phi_1 : ('F,'G)\text{nattransf}) = \lambda{:}\alpha.\ (\phi_2[:\alpha:]) \circ (\phi_1[:\alpha:])$
- Composing natural transformations with functors:
  $(\phi : ('F,'G)\text{nattransf}) \circ (H : 'H\ \text{functor}) = \lambda{:}\alpha.\ \phi[:\alpha\ 'H:]$
  $(H : 'H\ \text{functor}) \circ (\phi : ('F,'G)\text{nattransf}) = \lambda{:}\alpha.\ H\ (\phi\ [:\alpha:])$

- These definitions overload the $\circ$ composition operator.
  All these compositions yield natural transformations.

- In HOL2P:
  $\vdash$ *nattransf* $\phi\ F\ G \wedge$ *functor* $H \Longrightarrow$
  `TYINST` $((\theta_1 \mapsto \lambda\alpha.\ ((\alpha)\theta_1)\theta_3)\ (\theta_2 \mapsto \lambda\alpha.\ ((\alpha)\theta_2)\theta_3))$
  nattransf $(\lambda{:}\alpha.\ H\ \phi)\ (\lambda{:}\alpha\ \beta.\ H \circ F)\ (\lambda{:}\alpha\ \beta.\ H \circ G)$
- In HOL-Omega:
  $\vdash$ *nattransf* $\phi\ F\ G \wedge$ *functor* $H \Longrightarrow$
  nattransf $(H \circ \phi)\ (H \circ F)\ (H \circ G)$

Application: monads

- Monad type abbreviations:

  unit $= \lambda(\text{'}M :\mathbf{ty}\Rightarrow\mathbf{ty}).\ \forall\alpha.\ \alpha \to \alpha\ \text{'}M$

  bind $= \lambda(\text{'}M :\mathbf{ty}\Rightarrow\mathbf{ty}).\ \forall\alpha\ \beta.\ \alpha\ \text{'}M \to (\alpha \to \beta\ \text{'}M) \to \beta\ \text{'}M$

- Monad predicate, defined by three laws:

  monad (*unit* : *'M* unit, >>= : *'M* bind) =

  $(\forall:\alpha\ \beta.\ \forall(a:\alpha)\ (k:\alpha \to \beta\ M).$ (*left unit*)

  $\quad unit\ a >>= k = k\ a) \quad \wedge$

  $(\forall:\alpha.\ \forall(m:\alpha\ M).$ (*right unit*)

  $\quad m >>= unit = m) \quad \wedge$

  $(\forall:\alpha\ \beta\ \gamma.\ \forall(m:\alpha\ M)\ (k:\alpha\to\beta\ M)\ (h:\beta\to\gamma\ M).$ (*associativity*)

  $\quad (m >>= k) >>= h = m >>= (\lambda a.\ k\ a >>= h))$

- Example: The state monad:

  state $= \lambda\sigma\ \alpha.\ \sigma \to \alpha \times \sigma$ (*remember that* (σ, α)state $= \alpha\ (\sigma$ state$)$)

  state_unit $= \lambda:\alpha.\ \lambda(x:\alpha)\ (s:\sigma).\ (x, s)$

  state_bind $= \lambda:\alpha\ \beta.\ \lambda(w:(\sigma, \alpha)\text{state})\ (f:\alpha \to (\sigma, \beta)\text{state})\ (s:\sigma).$

  let $(x, s') = w\ s$ **in** $f\ x\ s'$

  HOL-Omega: |– *monad* (*state_unit* : $(\sigma$ state$)$unit, *state_bind* : $(\sigma$ state$)$bind)

Alternative definition of monads

- Alternative definition of monads based on 7 laws:

  map = $\lambda(\text{'}M :\mathbf{ty}\Rightarrow\mathbf{ty}).\ \forall\alpha\ \beta.\ (\alpha \to \beta) \to (\alpha\ \text{'}M \to \beta\ \text{'}M)$

  join = $\lambda(\text{'}M :\mathbf{ty}\Rightarrow\mathbf{ty}).\ \forall\alpha.\ (\alpha\ \text{'}M)\text{'}M \to \alpha\ \text{'}M$

  umj_monad (*unit* : *'M* unit, *map* : *'M* map, *join* : *'M* join) =

  1) $\forall:\alpha.$ — $map\ (\mathrm{I}:\alpha \to \alpha) = (\mathrm{I}:\alpha\ \text{'}M \to \alpha\ \text{'}M)$
  2) $\forall:\alpha\ \beta\ \gamma.\ \forall(f:\alpha \to \beta)\ (g:\beta \to \gamma).$ — $map\ (g \circ f) = map\ g \circ map\ f$
  3) $\forall:\alpha\ \beta.\ \forall(f:\alpha \to \beta).$ — $map\ f \circ unit = unit \circ f$
  4) $\forall:\alpha\ \beta.\ \forall(f:\alpha \to \beta).$ — $map\ f \circ join = join \circ map\ (map\ f)$
  5) $\forall:\alpha.$ — $join \circ unit = (\mathrm{I}:\alpha\ \text{'}M \to \alpha\ \text{'}M)$
  6) $\forall:\alpha.$ — $join \circ map\ unit = (\mathrm{I}:\alpha\ \text{'}M \to \alpha\ \text{'}M)$
  7) $\forall:\alpha.$ — $join \circ map\ join = join \circ join$

- Define *map, join* operators based on *unit,* >>= , or >>= based on *map, join*:

  `MMAP` (*unit,* >>=) = $\lambda:\alpha\ \beta.\ \lambda(f : \alpha \to \beta)\ (m : \alpha\ \text{'}M).\ m >>= (\lambda a.\ unit\ (f\ a))$

  `JOIN` (*unit,* >>=) = $\lambda:\alpha.\ \ \lambda(z : (\alpha\ \text{'}M)\text{'}M).\ z >>= \mathrm{I}$

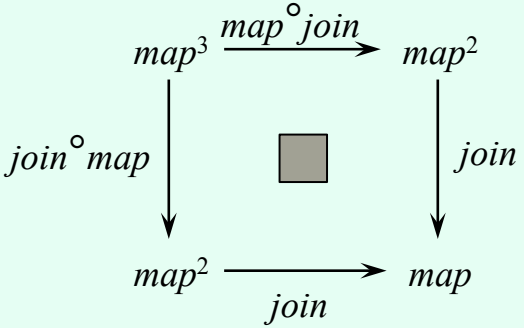
  `BIND` (*map, join*) = $\lambda:\alpha\ \beta.\ \lambda(m : \alpha\ \text{'}M)\ (k : \alpha \to \beta\ \text{'}M).\ join\ (map\ k\ m)$

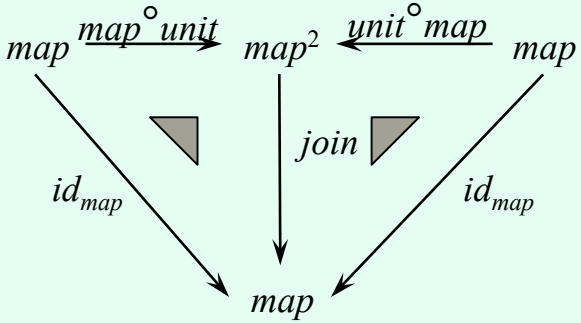
- Proved in HOL-Omega that these two monad definitions are equivalent.

Monads defined in category theory

- Third definition: *cat_monad*(*map, join, unit*) iff
  - *map* is a functor of type *'M* functor
  - *join* is a natural transformation between functors
    - *map* ° *map* of type (*'M* o *'M*)functor and
    - *map* of type *'M* functor,
    - where *join* has type (*'M* o *'M*, *'M*)nattransf, and
  - *unit* is a natural transformation between functors
    - *id* of type I functor and
    - *map* of type *'M* functor,
    - where *unit* has type (I, *'M*)nattransf
- such that the diagrams commute, i.e., these hold:
  - $join \circ (map \circ join) = join \circ (join \circ map)$
  - $join \circ (map \circ unit) = id_{map}$
  - $join \circ (unit \circ map) = id_{map}$
- where ° is composition between functors and/or natural transformations, as each situation requires (overloaded).
- This has been proven equivalent to the 7-law definition of monads (incl. types), so all three definitions of monads are equivalent.

Diagrams in Functor Category





Implementation

- The HOL-Omega theorem prover publicly released in 2009, now upgraded together with HOL4 version Kananaskis-6
- Backwards compatible with the HOL4 theorem prover
- Both kernels (de Bruijn indicies and name-carrying) upgraded
- Builds using either Moscow ML or Poly/ML
- Upgraded tools:
  - Rewriting (normal and higher-order)
  - Simplification (including type beta reduction of terms)
  - Definition of (mutual, nested, recursive) datatypes and records
  - Definition of function on such datatypes and records
- Not upgraded yet: rule induction definitions
- Most difficult part: correct integration of h.o. matching for terms, types

Status

- Implemented and working
  - Virtually completely backwards compatible with HOL4
  - Some facilities not yet upgraded to work with new types/terms
  - Type inference incomplete, but needed user annotations seem reasonable.
- Seems good platform to study category theory, e.g. monads
- Used by Jeremy Dawson of the Australian National University
  - Modeled a generalized version of monads
- Existential Types and Packages for data abstraction
- Current documentation is really scarce
  - for now, see 2009 paper and examples in <homedir>/examples/HolOmega
- Hope to soon put out a tutorial for new users
- More information from author's site:
  - http://www.trustworthytools.com
- Available for download from SourceForge :
  - svn checkout https://hol.svn.sf.net/svnroot/hol/branches/HOL-Omega

Conclusions

- The HOL-Omega logic adds significant power over higher order logic
  - Adds abstraction of type variables over terms
  - Adds abstraction of type variables over types
- Additional power infrequently used in practice, but when needed, it is absolutely required
- Type inference incomplete, but user annotations reasonable in practice
- Paper describes abstract syntax, set-theory semantics, axioms and rules of inference beyond HOL, examples of use, and implementation
- Implemented in a theorem prover tool, as a backwards-compatible extension of the Higher Order Logic HOL4 theorem prover
- Available for download from SourceForge or author's site:
  - http://www.trustworthytools.com
- Still experimental and under development but currently useful
- Supports mechanizing a substantial collection of new problems


The End
Soli Deo Gloria.