Basic Foundations of Isabelle/HOL

Peter Wullinger

May 16th 2007

HOL

- Higher Order Logic (HOL)

→ Reasoning over higher order constructs (e.g. functions of propositions)

- meta-logic: reasoning about logic
- At the core: typed λ calculus

Type Theory?

- Each symbol has a type
- A type is a (non-empty) set of individuals

→ Reasoning with types

Simple Type Theory

- Formal calculus available: Typed λ calculus
- Reason over types

$\leftarrow$ Functions have types, too!

$\rightarrow$ Higher order reasoning

- tλc is strongly normalizing for simple type theory!

Basic Syntactical Components

(using Paulsen's notation)

$\tau ::=$	(τ)	
	$\mathit{bool} \mid \mathit{nat} \mid \ldots$	basic types
	$\tau\ \mathit{list}$	(lists of τ)
	$\tau \mapsto \tau$	total functions
	$\tau \times \tau$	pairs (ASCII: $*$) (cross product)
	$'a \mid 'b \mid \ldots$	type variables
	$\ldots$	user defined types

We will explain all of this now.

Basic Syntactical Components

(using Paulsen's notation)

$\tau ::=$	(τ)	
	bool | *nat* | ...	basic types
	τ *list*	(lists of τ)
	$\tau \mapsto \tau$	total functions
	$\tau \times \tau$	pairs (ASCII: $*$) (cross product)
	'a|*'b*|...	type variables
	...	user defined types

Basic Types

bool $bool = \{\top, \bot\}$

this is the type of formulæ

nat natural numbers

note that 0, 1, . . . are overloaded.

when in doubt use $0 :: nat$.

list

simple recursive datatype:

$[]$	empty list
$x \# xs$	concatentation

Where $\#$ is of type $(list * list) \mapsto list$

Basic Syntactical Components

(using Paulsen's notation)

$\tau ::=$	(τ)	
	$bool \mid nat \mid \ldots$	basic types
	$\tau\ list$	(lists of τ)
	$\tau \mapsto \tau$	total functions
	$\tau \times \tau$	pairs (ASCII: $*$) (cross product)
	$'a \mid 'b \mid \ldots$	type variables
	$\ldots$	user defined types

Functions

- $Q : \beta$
- $x : \alpha$
- $\Phi = \lambda$ abstraction: $\lambda x : Q$

$\rightarrow$ *Phi* : $\alpha \mapsto \beta$

- Functions are types like all other types!

Basic Syntactical Components

(using Paulsen's notation)

$\tau ::=$	(τ)	
	$bool \mid nat \mid \ldots$	basic types
	$\tau\ list$	(lists of τ)
	$\tau \mapsto \tau$	total functions
	$\tau \times \tau$	pairs (ASCII: $*$) (cross product)
	$'a \mid 'b \mid \ldots$	type variables
	$\ldots$	user defined types

Products

- tλc extension of Isabelle/HOL:

$$\frac{a : \alpha \qquad b : \beta}{< a, b >: (\alpha \times \beta)}$$

- *left injection* and *right injection* possible:
- Insert *left* or *right* parameter
- Currying can in most cases achieve the same

$\rightarrow$ Avoid and use with care

Basic Syntactical Components

(using Paulsen's notation)

$\tau ::=$	(τ)	
	$bool \mid nat \mid \ldots$	basic types
	$\tau\ list$	(lists of τ)
	$\tau \mapsto \tau$	total functions
	$\tau \times \tau$	pairs (ASCII: $*$) (cross product)
	$'a \mid 'b \mid \ldots$	type variables
	$\ldots$	user defined types

Type Variables

- These can be used to implement polymorphism
- Typically used for user defined

$\rightarrow$ *'a list* a list of *'a* (whatever that is)

- Isabelle performs *lazy* type checking

Basic Syntactical Components

(using Paulsen's notation)

$\tau ::=$	(τ)	
	$bool \mid nat \mid \ldots$	basic types
	$\tau\ list$	(lists of τ)
	$\tau \mapsto \tau$	total functions
	$\tau \times \tau$	pairs (ASCII: $*$)
		(cross product)
	$'a \mid 'b \mid \ldots$	type variables
	$\ldots$	user defined types

User Defined Types

- Recursive definition of new types
- Also: Arbitrary new types

- Restriction:
  - May not be empty (this is proved on definition)
- Example:

```
datatype 'a mylist =
   Nil
   | Cons 'a ('a mylist)
;
```

Terms

$$\begin{array}{rcccc} term & ::= & (term) & | & a \\ & | & \lambda x.term & | & term\ term \\ & | & \ldots & & \end{array}$$

Terms

$$\begin{array}{rcccc} \textit{term} & ::= & (\textit{term}) & | & a \\ & | & \lambda x.\textit{term} & | & \textit{term}\ \textit{term} \\ & | & \ldots & & \end{array}$$

Abstraction

- $\Phi \equiv \lambda x.P$
- $x : \alpha, P : \beta \Rightarrow \Phi : (\alpha \mapsto \beta)$
- creates a function mapping

Terms

$$\begin{array}{rcccc} term & ::= & (term) & | & a \\ & | & \lambda x.term & | & term\ term \\ & | & \ldots & & \end{array}$$

Application

- The inverse of abstraction
- $\Phi \equiv (P(x)a)$ or $(\lambda x.P)a$
- $P : (\alpha \mapsto \beta), a : \alpha \Rightarrow \Phi : \beta$

Conversions

α-conversion

β-conversion

η-conversion

Conversions

α-conversion $\lambda y(\lambda x.x)y) \equiv_\alpha \lambda z.(\lambda x.x)z$
bound variable names don't matter

β-conversion

η-conversion

Conversions

α-conversion $\lambda y(\lambda x.x)y) \equiv_\alpha \lambda z.(\lambda x.x)z$
bound variable names don't matter

β-conversion $(\lambda x.P)a \equiv_\beta P[x \leftarrow a]$
function application (β-reduction)

η-conversion

Conversions

α-conversion $\lambda y(\lambda x.x)y) \equiv_\alpha \lambda z.(\lambda x.x)z$
bound variable names don't matter

β-conversion $(\lambda x.P)a \equiv_\beta P[x \leftarrow a]$
function application (β-reduction)

η-conversion $M \equiv_\eta \lambda x.(Mx)$ if x is not free in M
normal form analysis

Conversions

α-conversion $\lambda y(\lambda x.x)y) \equiv_\alpha \lambda z.(\lambda x.x)z$
bound variable names don't matter

β-conversion $(\lambda x.P)a \equiv_\beta P[x \leftarrow a]$
function application (β-reduction)

η-conversion $M \equiv_\eta \lambda x.(Mx)$ if x is not free in M
normal form analysis

Normal forms

- these are strongly normalizing

$\rightarrow$ any application sequence terminates in a normal form

- this normal form is unique

Formulæ

$$
\begin{array}{lcccccc}
\mathit{form} & ::= & (\mathit{form}) & | & \mathit{term} = \mathit{term} & | & \neg \mathit{form} \\
 & | & \mathit{form} \wedge \mathit{form} & | & \mathit{form} \vee \mathit{form} & | & \mathit{form} \rightarrow \mathit{form} \\
 & | & \forall x.\mathit{form} & | & \exists x.\mathit{form} & &
\end{array}
$$

Formulæ

$$\begin{array}{rcccccc} form & ::= & (form) & | & term = term & | & \neg form \\ & | & form \wedge form & | & form \vee form & | & form \rightarrow form \\ & | & \forall x.form & | & \exists x.form & & \end{array}$$

Quantifiers

- without quantifiers, the logic cannot be higher order
- only one of $\forall$ or $\exists$ is needed
- $\forall$ is written $\bigwedge$ in the meta-logic (! !)
- formally:

$$(\bigwedge x.(\lambda x : \alpha.P : bool)x) : bool$$

Formulæ

$$
\begin{array}{rcccccc}
\mathit{form} & ::= & (\mathit{form}) & | & \mathit{term} = \mathit{term} & | & \neg \mathit{form} \\
 & | & \mathit{form} \wedge \mathit{form} & | & \mathit{form} \vee \mathit{form} & | & \mathit{form} \rightarrow \mathit{form} \\
 & | & \forall x.\mathit{form} & | & \exists x.\mathit{form} & &
\end{array}
$$

Quantifiers

- Introduction and Elimination rules for quantifiers
- extended tλc is still strongly normalising

Rewrite Rules

- Isabelle has rewrite operations defined on Isabelle Pure Syntax
- Defining logics, one needs to add rewriting rules
- Pure Syntax looks very similar to typed λ calculus
- We introduct differences when encountered

Isabelle Pure Rewrite Rules

- There are a few of these
- Most of them are defined using macros or translation functions

e.g. $(\%x.\ P)a ==> P[x \leftarrow a]$

where $P[x \leftarrow a]$ is the result of (built-in) substitution.

- For now, we just assume that $\alpha\beta\eta$ translations are done automatically without presenting rules.

HOL rules

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

- These are the basic rewriting rules.
- They turn basic HOL formalæ into Isabelle Pure (tλc)

HOL rules

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

It is always possible to introduce/delete type variables.

HOL rules

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

Equal terms can be subsituted and imply proving equality

HOL rules

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

Two functional abstractions are equal if they return the same result for all parameters (similar to η conversion)

HOL rules

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

Object logic implication maps to higher order implication

HOL rules

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

Syntax Note: $[\|\ldots|]$ denotes grouping of presumptions for natural deduction. For rule application, this intuitively means *from the list of things that are currently true, select something matching and unify*.

That is modus ponens can be applied if the presumption of a meta-level implication is also true.

HOL rules

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

Two formulæ imply each other if only if they are equal.

HOL rules

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

Syntax Note: @ is Hilbert's ϵ-operator. $@x.\ P\ x$ selects an item x that satisfies P.

This is the transfinite axiom, that is if $P(x)$ is true (and thus x is free), then there is also some x that satisfies $P(x)$.

HOL rules

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

This is the classical axiom of choice and gives us *tertium non datur*, i.e. classical logics.

HOL definitions

$$
\begin{array}{lll}
\mathit{True\_def} & \mathit{True} & == \quad ((\%x :: \mathit{bool}.\ x) = (\%x.\ x)) \\
\mathit{All\_def} & \mathit{All} & == \quad (\%P.\ P = (\%x.\ \mathit{True}) \\
\mathit{Ex\_def} & \mathit{Ex} & == \quad (\%P.\ P(@x.\ P\ x)) \\
\ldots & & \\
\mathit{And\_def} & \mathit{op\&} & == \quad \%PQ\ .!R.\ (P --> Q --> R) --> R)
\end{array}
$$

HOL definitions

$$
\begin{array}{lll}
\mathit{True\_def} & \mathit{True} & == \quad (\%x :: \mathit{bool}.\ x) = (\%x.\ x)) \\
\mathit{All\_def} & \mathit{All} & == \quad (\%P.\ P = (\%x.\ \mathit{True}) \\
\mathit{Ex\_def} & \mathit{Ex} & == \quad (\%P.\ P(@x.\ P\ x)) \\
\ldots & & \\
\mathit{And\_def} & \mathit{op\&} & == \quad \%PQ\ .!R.\ (P --> Q --> R) --> R)
\end{array}
$$

Equal terms can be reduced to *True*

HOL definitions

$$
\begin{array}{lll}
\mathit{True\_def} & \mathit{True} & == \quad ((\%x :: \mathit{bool}.\ x) = (\%x.\ x)) \\
\mathit{All\_def} & \mathit{All} & == \quad (\%P.\ P = (\%x.\ \mathit{True}) \\
\mathit{Ex\_def} & \mathit{Ex} & == \quad (\%P.\ P(@x.\ P\ x)) \\
\ldots & & \\
\mathit{And\_def} & \mathit{op\&} & == \quad \%PQ\ .!R.\ (P --> Q --> R) --> R)
\end{array}
$$

- This is a little bit harder
- The all-quantor takes a proposition of some function type ($\alpha \mapsto \mathit{bool}$, where $x : \alpha$) as argument
- and returns if the given parametrized proposition can be simplified to true.

HOL definitions

$$
\begin{array}{lll}
\mathit{True\_def} & \mathit{True} & == \quad ((\%x :: \mathit{bool}.\ x) = (\%x.\ x)) \\
\mathit{All\_def} & \mathit{All} & == \quad (\%P.\ P = (\%x.\ \mathit{True}) \\
\mathit{Ex\_def} & \mathit{Ex} & == \quad (\%P.\ P(@x.\ P\ x)) \\
\ldots & & \\
\mathit{And\_def} & \mathit{op\&} & == \quad \%PQ\ .!R.\ (P --> Q --> R) --> R)
\end{array}
$$

- Similar to the all-quantor
- A predicate fulfill the existential quantor if there exist's some x fulfilling it.
- Rewritten using the ϵ operator

HOL definitions

$$
\begin{array}{lll}
\mathit{True\_def} & \mathit{True} & == \quad (\%x :: \mathit{bool}.\ x) = (\%x.\ x)) \\
\mathit{All\_def} & \mathit{All} & == \quad (\%P.\ P = (\%x.\ \mathit{True}) \\
\mathit{Ex\_def} & \mathit{Ex} & == \quad (\%P.\ P(@x.\ P\ x)) \\
\ldots & & \\
\mathit{And\_def} & \mathit{op\&} & == \quad \%PQ\ .!R.\ (P --> Q --> R) --> R)
\end{array}
$$

We skip all the others here and just show another interesting one.

HOL definitions

$$
\begin{array}{lll}
\mathit{True\_def} & \mathit{True} & == \quad (\%x :: \mathit{bool}.\ x) = (\%x.\ x)) \\
\mathit{All\_def} & \mathit{All} & == \quad (\%P.\ P = (\%x.\ \mathit{True}) \\
\mathit{Ex\_def} & \mathit{Ex} & == \quad (\%P.\ P(@x.\ P\ x)) \\
\ldots & & \\
\mathit{And\_def} & \mathit{op\&} & == \quad \%PQ\ .!R.\ (P --> Q --> R) --> R)
\end{array}
$$

- & (and, $\wedge$) is rewritten into object level implication.

$\rightarrow$ If $P \wedge Q$, both implications only depend on R. This assumption is then discharged since $R \rightarrow R$.

$\leftarrow$ If one of P or Q is false, there is an $R = \mathit{False}$ and thus the last implication is false.

Derived Rules

- The above rules are the core rules
- HOL comes with a set of derived rules
- These are easier to read

e.g. conjE $[\!|\ P\&Q\ |\!];\ [\!|\ P;\ Q\ |\!] \Longrightarrow R\ |\!] \Longrightarrow R$

Subgoal

Consider the following subgoal:

$$\bigwedge a\ .\ \mathit{occurs}\ a\ ([] \mathbin{@} \mathit{ys}) = \mathit{occurs}\ a\ [] + \mathit{occurs}\ a\ \mathit{ys}$$

We introduce rules as we go.

Proof

$$\bigwedge a\ .\ \mathit{occurs}\ a\ ([]\ @\ \mathit{ys})$$
$$=\ \mathit{occurs}\ a\ []\ +\ \mathit{occurs}\ a\ \mathit{ys}$$

Proof

$$\bigwedge a\ .\ occurs\ a\ ([]\ @\ ys) = occurs\ a\ []\ +\ occurs\ a\ ys$$

$$[]\ @\ ?ys \Leftarrow ?ys \qquad \bigwedge a\ .\ occurs\ a\ ys = occurs\ a\ []\ +\ occurs\ a\ ys$$

Proof

$$\bigwedge a\ .\ occurs\ a\ ([]\ @\ ys) = occurs\ a\ []\ +\ occurs\ a\ ys$$

$$[]\ @\ ?ys \Leftarrow ?ys \qquad \bigwedge a\ .\ occurs\ a\ ys = occurs\ a\ []\ +\ occurs\ a\ ys$$

$$occurs\ ?a\ [] \Leftarrow 0 \qquad \bigwedge a\ .\ occurs\ a\ ys = 0 + occurs\ a\ ys$$

Example proof

Proof

$$\bigwedge a\ .\ occurs\ a\ ([]\ @\ ys) = occurs\ a\ []\ +\ occurs\ a\ ys$$

$$[]\ @\ ?ys \Leftarrow ?ys \qquad \bigwedge a\ .\ occurs\ a\ ys = occurs\ a\ []\ +\ occurs\ a\ ys$$

$$occurs\ ?a\ [] \Leftarrow 0 \qquad \bigwedge a\ .\ occurs\ a\ ys = 0 + occurs\ a\ ys$$

$$0\ +\ ?x \Leftarrow ?x \qquad \bigwedge a\ .\ occurs\ a\ ys = occurs\ a\ ys$$

Proof

	$\bigwedge a\,.\ occurs\ a\ ([] \mathbin{@} ys)$ $=\ occurs\ a\ [] + occurs\ a\ ys$
$[] \mathbin{@} ?ys \Leftarrow ?ys$	$\bigwedge a\,.\ occurs\ a\ ys = occurs\ a\ [] + occurs\ a\ ys$
$occurs\ ?a\ [] \Leftarrow 0$	$\bigwedge a\,.\ occurs\ a\ ys = 0 + occurs\ a\ ys$
$0 + ?x \Leftarrow ?x$	$\bigwedge a\,.\ occurs\ a\ ys = occurs\ a\ ys$
$?xs = ?xs \Leftarrow True$	$\bigwedge a\,.\ True$

Proof

	$\bigwedge a\,.\ occurs\ a\ ([] \ @\ ys)$ $=\ occurs\ a\ []\ +\ occurs\ a\ ys$
$[] \ @\ ?ys \Leftarrow ?ys$	$\bigwedge a\,.\ occurs\ a\ ys\ =\ occurs\ a\ []\ +\ occurs\ a\ ys$
$occurs\ ?a\ [] \Leftarrow 0$	$\bigwedge a\,.\ occurs\ a\ ys\ =\ 0 +\ occurs\ a\ ys$
$0\ +\ ?x \Leftarrow ?x$	$\bigwedge a\,.\ occurs\ a\ ys\ =\ occurs\ a\ ys$
$?xs\ =\ ?xs \Leftarrow True$	$\bigwedge a\,.\ True$
$\bigwedge a\ P \Leftarrow P$	$True$

Proof

	$\bigwedge a\,.\ occurs\ a\ ([]\ @\ ys)$ $=\ occurs\ a\ []\ +\ occurs\ a\ ys$
$[]\ @\ ?ys\ \Leftarrow\ ?ys$	$\bigwedge a\,.\ occurs\ a\ ys\ =\ occurs\ a\ []\ +\ occurs\ a\ ys$
$occurs\ ?a\ []\ \Leftarrow\ 0$	$\bigwedge a\,.\ occurs\ a\ ys\ =\ 0\ +\ occurs\ a\ ys$
$0\ +\ ?x\ \Leftarrow\ ?x$	$\bigwedge a\,.\ occurs\ a\ ys\ =\ occurs\ a\ ys$
$?xs\ =\ ?xs\ \Leftarrow\ True$	$\bigwedge a\,.\ True$
$\bigwedge a\ P\ \Leftarrow\ P$	$True$

Now this lemma can be used as a new rule!

Try this in Isabelle

That's it for today

Try to prove the above in Isabelle