Church’s Type Theory
Ordered Pairs

The HOL system has an operator for constructing ordered pairs, which is called \( ; \):

It is curried by itself, so that \((a, b)\) is represented by

\[
(\ , \cdot a) \cdot b.
\]

For the pairing operator, a new type constructor has to be introduced:

If \(a : A\), and \(b : B\), then \(a, b\) has type \(A \times B\).
Polymorphism/Type Variables

HOL has built-in type Bool. All other types are user defined, or defined in libraries.

In addition, there are type variables, which enable polymorphism.

For example, equality has type $\alpha \rightarrow \alpha \rightarrow \text{Bool}$. 

In concrete applications of $\approx$, $\alpha$ can be instantiated to a concrete type.

For example, in $1 + 1 \approx 2$, it will be instantiated to Int.

The pair constructor has type $\alpha \rightarrow \beta \rightarrow (\alpha \times \beta)$. In the concrete case of $(4, \mathbf{T})$, it has type $\text{Int} \rightarrow \text{Bool} \rightarrow (\text{Int} \times \text{Bool})$. 
Hol Terms

We give a formal definition of the terms of HOL. Note that formulas and terms are not distinguished. A formula is simply a term that has type \( \text{Bool} \).

**Definition:** The terms of HOL are recursively defined as follows:

- A variable is a term.
- If \( f \) and \( t \) are terms, then \( f \cdot t \) is a term.
- If \( x \) is a variable, and \( t \) is a term, then \( \lambda x \ t \) is a term.
- If \( t_1, t_2 \) are terms, then \( t_1, t_2 \) is a term.
HOL Types

Definition: The types of HOL are recursively defined as follows:

- A type variable is a type.
- If $A$ and $B$ are types, then $A \to B$ is a type.
- If $A$ and $B$ are types, then $A \times B$ is a type.
Sequents

The HOL logic is based on sequents.

Definition: A sequent is an object of form $A_1, \ldots, A_p \vdash B$.

The meaning is: $A_1, \ldots, A_p$ imply $B$. 
Type Checking of Sequents

In a sequent, all $A_1, \ldots, A_p$, and $B$ must have type $\text{Bool}$.

Type checking is done automatically, and the algorithm uses unification to instantiate type variables, when necessary.

1. Look up the types for identifiers that have a definition. If a defined identifier is polymorphic (has variables in its type), then each occurrence receives a fresh set of variables.

2. Assign to every undeclared identifier a fresh type variable $\alpha_i$. When an undeclared variable occurs more than once, the different occurrences receive the same $\alpha_i$.

3. Typecheck the sequent and unify type variables if necessary.

When finished, the algorithm either has constructed a unifying substitution $\Theta$, or it has failed.
Examples of Type Checking

We check the sequent $a \approx b \vdash b \approx c$. First write it in Curried form:

$$(\approx \ a \ b) \vdash (\approx \ b \ c).$$

Look up the types of the symbols:

$$((\approx: \alpha \to \alpha \to \text{Bool}) \ (a: \beta) \ (b: \gamma)) \vdash ((\approx: \delta \to \delta \to \text{Bool}) \ (b: \gamma) \ (c: \zeta)).$$

Then unify $\alpha \equiv \beta$, $\gamma \equiv \beta$, $\delta \equiv \beta$, $\zeta \equiv \beta$. 
Examples of Type Checking (2)

We check the sequent \( \vdash f(a) \approx f(4) \). First write it in the form

\[
\vdash (\approx (f\ a) (f\ 4)).
\]

Look up the types of the symbols:

\[
\vdash ( (\approx: \alpha \rightarrow \alpha \rightarrow \text{Bool}) ( (f: \beta) (a: \gamma) ) ( (f: \beta) (4: \text{Int}) ) ).
\]

The identifier \( f: \beta \) is applied on \( a: \gamma \). This means that \( \beta \) must have form \( \beta_1 \rightarrow \beta_2 \) and that \( \beta_1 = \gamma \). Since \( \approx \) has type \( \alpha \rightarrow \alpha \rightarrow \text{Bool} \), we see that \( \alpha = \beta_2 \).

The result is

\[
\vdash ( (\approx: \alpha \rightarrow \alpha \rightarrow \text{Bool}) ( (f: \gamma \rightarrow \alpha) (a: \gamma) ) ( (f: \gamma \rightarrow \alpha) (4: \text{Int}) ) ).
\]

Looking at the second term, we see that \( \gamma = \text{Int} \). The sequent is accepted, and the variable \( \alpha \) remains open.
Theorems/Definitions

HOL has some data structure that stores proven theorems. There is no distinction between definitions and theorems. If $c$ is an unused variable, and $t$ is a term without free (term or type) variables, then it is possible to add the theorem

$$\vdash c \approx t.$$
Type Definitions
Primitive Inference Rules for Sequents

Basic Equality Rules:

\[ \text{REFL} \quad \frac{}{\Gamma \vdash t \approx t} \]

\[ \text{TRANS} \quad \frac{\Gamma \vdash s \approx t, \Delta \vdash t \approx u}{\Gamma \cup \Delta \vdash s \approx u} \]

\[ \text{MK_COMB} \quad \frac{\Gamma \vdash s \approx t, \Delta \vdash u \approx v}{\Gamma \cup \Delta \vdash (s \cdot u) \approx (t \cdot v)} \]
Rules involving $\lambda$

If $x$ is not free in an assumption in $\Gamma$, then

$$\frac{\Gamma \vdash s \approx t}{\Gamma \vdash (\lambda x.s) \approx (\lambda x.t)}$$

$$\frac{\Gamma \vdash ((\lambda x.t) \cdot x) \approx t}{\text{BETA}}$$

Note that this is a very restricted form of $\beta$-equivalence. We will later see that all other forms are derivable.
Logical Rules

ASSUME \[ p \vdash p \]

EQ_MP \[ \Gamma \vdash p \approx q \quad \Delta \vdash p \]
\[ \Gamma \cup \Delta \vdash q \]

DEDUCT_ANTISYM_RULE \[ \Gamma \vdash p \quad \Delta \vdash q \]
\[ (\Gamma \setminus \{ q \}) \cup (\Delta \setminus \{ p \}) \vdash p \approx q \]
**Instantiation Rules**

If $x$ is an identifier that does not occur in a theorem, then

\[
\text{INST} \quad \frac{\Gamma \vdash A}{\Gamma[x := t] \vdash A[x := t]}
\]

If $\alpha$ is a type variable, then

\[
\text{INST\_TYPE} \quad \frac{\Gamma \vdash A}{\Gamma[\alpha := T] \vdash A[\alpha := T]}
\]
Definitions of Logical Operators

\( \top = (\lambda x. x) \approx (\lambda x. x) \),
\( \land = \lambda p. \lambda q. \lambda f. (f \ p \ q) \approx \lambda f. (f \ \top \ \top) \),
\( \rightarrow \) (the logical operator) = \( \lambda p. \lambda q. (p \land q) \approx p \),
\( \forall = \lambda P. (P \approx \lambda Q. \top) \),
\( \exists = \lambda P. \forall Q. (\forall x. (P \ x)) \rightarrow Q \) \rightarrow Q,
\( \lor = \lambda P. \lambda Q. \forall R. (P \rightarrow R) \rightarrow (Q \rightarrow R) \rightarrow R \),
\( \bot = \forall P. P \),
\( \neg = \lambda P. (P \rightarrow \bot) \),
\( \exists! = \lambda P. (\exists P) \land \forall x. y. (P \ x) \land (P \ y) \rightarrow (x \approx y) \),
Axioms of HOL

Extensionality:

\[ \vdash (\lambda x. (t \; x)) \approx t. \]

Epsilon: (Hilbert choice operator)

\[ \vdash \forall x. ((P \; x) \to (P \; (\epsilon (\lambda x. (P \; x))))). \]

There is one declared type Ind.

There are infinitely many individuals:

\[ \vdash \exists f: \text{Ind} \to \text{Ind}. \; (\forall x_1. x_2. \; (f \; x_1) \approx (f \; x_2) \to x_1 \approx x_2) \land \neg ((\forall y. \exists x. \; y \approx (f \; x)). \]
Some Info about the HOL System

HOL is an interactive system written in ocaml. The user proofs sequents by typing ocaml commands.

HOL provides a type theorem which consists of true sequents. The constructors of type theorem correspond to the deduction rules given before. As a consequence, every sequent that is a theorem, has a proof.

The primitive deduction rules of HOL are pretty unpleasant, and their implementations are even more unpleasant.

Fortunately, the user does not have to deal directly with the primitive inference rules. HOL contains a lot of additional functions that automatically construct theorems. The user normally calls the additional functions.
Some Info about the HOL System

We see that a proof consists of a sequence of commands in Ocaml. A disadvantage of this procedural approach is that a proof contains almost no formulas, and that therefore, proofs are unreadable. A proof contains lines like

```
e( EVERY [ CONJ_TAC; ( REPEAT STRIP_TAC );
           ASM_REWRITE_TAC [ plusdef ] ] );;
```

which tells us completely nothing about the logical structure of the proof.

Of course, it is also an advantage that the system constructs the formulas, and typing formula is tedious.
Using the HOL System

One normally types commands in the Ocaml shell, and simultaneously stores them in a file, so that the complete proof can be replayed afterwards.