4 Higher-Order Logic

Desired for applications, e. g., in mathematics:

quantifications over functions and predicates,

functions and predicates applied to functions and predicates,

partially applied functions,

anonymous functions,

first-class booleans,

expressivity: define, e. g., "the" natural numbers, "the" reals.

Higher-order logic satisfies these needs.

4.1 The Starting Point: λ-Calculus

Untyped λ-calculus (Church 1930).

Syntax:

Terms:	$t ::= c$		(Constant)
	$\mid x$		(Variable)
	$\mid (t_1\ t_2)$		(Application)
	$\mid (\lambda x.\, t)$		(Abstraction)

Substitution:

$x\{x \mapsto s\} = s$.

$y\{x \mapsto s\} = y$ if $y \neq x$.

$c\{x \mapsto s\} = c$.

$(t_1\ t_2)\{x \mapsto s\} = (t_1\{x \mapsto s\}\ t_2\{x \mapsto s\})$.

$(\lambda x.\, t)\{x \mapsto s\} = (\lambda x.\, t)$.

$(\lambda y.\, t)\{x \mapsto s\} = (\lambda z.\, (t\{y \mapsto z\}\{x \mapsto s\}))$ if $y \neq x$, z fresh.

Conversion rules (to be applied to arbitrary subterms):

$t \to_\alpha t'$
if t and t' are equal upto renaming of bound variables.

$((\lambda x.\, t)\ s) \to_\beta t\{x \mapsto s\}$.

$(\lambda x.\,(t\ x)) \to_\eta t$
if x does not occur freely in t.

Properties of the untyped λ-calculus:

β-conversion may not terminate.

Works as a model of computation (Turing-complete).

But basing a logic on it leads to problems (similarly to Russell's paradox).

Solution: introduce types.

4.2 Typed λ-Calculus

Typed λ-calculus:

Developed by Church in 1940.

Also known as *Simple Type Theory*.

Note: Many variants (syntax and semantics).

Types

Types are defined recursively:

o is the type of Booleans, of order 0.

ι is the type of individuals, of order 1.

if τ_1 and τ_2 are types then $\tau_1 \to \tau_2$ is a type, of order $\max(\text{order}(\tau_1) + 1, \text{order}(\tau_2))$

We also write $\tau_1 \to \cdots \to \tau_n \to \tau$ or $\tau_1, \ldots, \tau_n \to \tau$ for $\tau_1 \to (\cdots \to (\tau_n \to \tau) \ldots)$.

Terms

Given a non-empty set of constants and a collection of non-empty sets of variables for each type,

constants are terms,

variables are terms,

if t_1 and t_2 are terms then $(t_1\ t_2)$ is a term,

if x is a variable and t is a term then $\lambda x.\,t$ is a term.

Types of Terms

Given a non-empty set S of typed constants and a collection of non-empty sets of variables for each type, the term t is of type

o if $t \in \{\top, \bot\}$,

τ if $t \in S$ has type τ,

τ if $t = x_{(\tau)}$ is a variable of type τ,

$\tau_1 \to \tau_2$ if $t = \lambda x_{(\tau_1)}.\, t_{1(\tau_2)}$,

τ_2 if $t = (t_{1(\tau_1 \to \tau_2)}\ t_{2(\tau_1)})$.

A term is well-typed if a type can be associated to it according to the previous definition. We only consider well-typed terms in what follows.

Normal Forms

For well-typed terms, we can define two kinds of normal forms:

$\beta\eta$-short normal form:

> Apply β forward exhaustively (terminates because of typing), then apply η forward exhaustively.

$\beta\eta$-long normal form:

> Apply β forward exhaustively (terminates because of typing), then apply η backward exhaustively (respecting the types).

4.3 Semantics

A well-founded formula is a term of type o.

How to evaluate the truth of such a formula?

Classical Models

Let D be a non-empty set, for each type τ we define the following collection, denoted as the *frame* of the type

the frame of $\tau = o$ is $[\![o, D]\!] = \{0, 1\}$

the frame of $\tau = \iota$ is $[\![\iota, D]\!] = D$

the frame of $\tau = \tau_1 \to \tau_2$ is $[\![\tau_1 \to \tau_2, D]\!]$, the collection of all functions mapping $[\![\tau_1, D]\!]$ into $[\![\tau_2, D]\!]$

A higher-order *classical model* is a structure $\mathcal{M} = \langle D, \mathcal{I} \rangle$ where D is a non-empty set called the *domain* of the model and $\mathcal{I}$ is the *interpretation* of the model, a mapping such that

if $c_{(\tau)}$ is a constant then $\mathcal{I}(c) \in [\![\tau, D]\!]$,

$\mathcal{I}(=_{(\tau \to \tau \to o)})$ is the equality relation on $[\![\tau, D]\!]$.

By adding an *assignment* function α such that for any variable $x_{(\tau)}$, $\alpha(x) \in [\![\tau, D]\!]$, it becomes possible to evaluate the truth value of higher-order formulas as in first-order logic.

The *evaluation* $\mathcal{V}_{\mathcal{M},\alpha}(t)$ of a term t given a model $\mathcal{M} = \langle D, \mathcal{I} \rangle$ and an assignment α is recursively defined as

$\mathcal{I}(c)$ if t is a constant c

$\alpha(x)$ if t is a variable x

the function from $[\![\tau_1, D]\!]$ to $[\![\tau_2, D]\!]$ that maps every $a \in [\![\tau_1, D]\!]$ to $\mathcal{V}_{\mathcal{M},\alpha[x \mapsto a]}(t) \in [\![\tau_2, D]\!]$ if $t = \lambda x_{(\tau_1)}.\, t_{(\tau_2)}$

$(\mathcal{V}_{\mathcal{M},\alpha}(t_1))(\mathcal{V}_{\mathcal{M},\alpha}(t_2))$ if $t = (t_{1(\tau_1 \to \tau_2)}\ t_{2(\tau_1)})$

Truth evaluation:

Given a model $\mathcal{M} = \langle D, \mathcal{I} \rangle$ and an assignment α, a well-founded formula F is true in $\mathcal{M}$ with respect to α, denoted as $\mathcal{M}, \alpha \models F$ iff $\mathcal{V}_{\mathcal{M},\alpha}(F) = 1$

F is satisfiable in $\mathcal{M}$ iff there exists an assignment α such that $\mathcal{M}, \alpha \models F$

F is valid in $\mathcal{M}$, denoted $\mathcal{M} \models F$ iff for all assignments α, $\mathcal{M}, \alpha \models F$

F is valid, denoted $\models F$ iff for all models $\mathcal{M}$, $\mathcal{M} \models F$

These notions extend straightforwardly to sets of formulas.

Problems with the Classical Semantics

HOL with classical semantics (cHOL) is very expressive, but:

- In FOL, every unsatisfiable set of formulas has a finite unsatisfiable subset. This is no longer the case in cHOL. (Loss of compactness)
- No proof procedure able to derive all consequences of a set of formulas can exist in cHOL. (Loss of strong completeness)
- No proof procedure able to derive all valid sets of formulas can exist in cHOL. (Loss of weak completeness)
- The status of validity of some formulas is unclear.

Henkin Semantics

To solve the previously mentioned issues, it is possible to generalize the notion of a model by relaxing the notion of a frame into that of a Henkin frame. Given a non-empty set D,

$[\![o, D]\!] = \{0, 1\}$

$[\![\iota, D]\!] = D$

$[\![\tau_1 \to \tau_2, D]\!]$ is the collection of ~~all~~ *some* functions mapping $[\![\tau_1, D]\!]$ into $[\![\tau_2, D]\!]$ *with some additional closure conditions*.

Henkin vs Classical Semantics

- Every classical model is a Henkin model, therefore every formula true in all Henkin models is true in all classical models.
- There are formulas true in all classical models that are not true in all Henkin models.
- There are (weak) complete proof procedures for HOL with Henkin semantics.

4.4 Higher-Order Term Unification

We consider unification modulo α, β, η.

In FOL, there exists a unique mgu for two unifiable terms.

This is no longer true in HOL.

For example, consider $t_1 = y\ x$ and $t_2 = c$ where x, y are variables and c is a constant. The unifiers of t_1 and t_2 are $\{y \mapsto \lambda z.\, c\}$ and $\{y \mapsto \lambda z.\, z,\ x \mapsto c\}$.

Some equations do not even have finite complete sets of unifiers, e. g., $f\ (x\ c) = x\ (f\ c)$ with f and c constants.

Even worse, the higher-order unification problem is undecidable.

Huet's Pre-Unification Algorithm

Given:

a unification problem E, i.e. a finite set of equations in $\beta\eta$-long normal form.

Goal:

find a substitution σ such that $E\sigma$ contains only syntactically equal equations.

Rigid and Flexible Terms

Every term can be written in the form $\lambda x_1 \ldots x_n.\, u_0\ u_1 \ldots u_k$ $(n \geq 0,\ k \geq 0)$, where u_0 is a constant or a bound or free variable.

u_0 is called the *head* of the term.

A term is called *rigid* if its head symbol is a constant or a bound variable. Otherwise its head symbol is a free variable and the term is called *flexible*.

Rigid-Rigid Equations

Two rules can be applied depending on the head symbols in the rigid-rigid equation.

Simplify:

$$\frac{E \cup \{\lambda x_1 \ldots x_n.\, f\ u_1 \ldots u_p \approx \lambda x_1 \ldots x_n.\, f\ v_1 \ldots v_p\}}{E \cup \{\lambda x_1 \ldots x_n.\, u_1 \approx \lambda x_1 \ldots x_n.\, v_1, \ldots, \lambda x_1 \ldots x_n.\, u_p \approx \lambda x_1 \ldots x_n.\, v_p\}}$$

where f is a constant or a bound variable.

Fail:

$$\frac{E \cup \{\lambda x_1 \ldots x_n.\, f\ u_1 \ldots u_p \approx \lambda x_1 \ldots x_n.\, g\ v_1 \ldots v_q\}}{\bot}$$

where f and g are distinct constants or bound variables.

Flexible-Rigid Equations

There is only one rule to handle flexible-rigid equations.

Generate:

$$\frac{E}{E\sigma}$$

where

$(\lambda x_1 \ldots x_n.\, z\ u_1 \ldots u_p \approx \lambda x_1 \ldots x_n.\, f\ v_1 \ldots v_q) \in E$,
z is a free variable, f is a constant or bound variable,
$h \in \{f, y_1, \ldots, y_p\}$ if f is a constant and $h \in \{y_1, \ldots, y_p\}$ otherwise,
$z_1, \ldots, z_r$ are fresh free variables,
$\sigma = \{z \mapsto \lambda y_1, \ldots, y_p.\, h\ (z_1\ y_1 \ldots y_p) \ldots (z_r\ y_1 \ldots y_p)\}$.

Flexible-Flexible Equations

The following result, also by Huet, handles flexible-flexible equations.

Proposition 4.1 *A unification problem E containing only flexible-flexible equations has always a solution.*

Proof. For every type τ let w_τ be a fixed fresh variable of type τ. Define θ as the substitution that maps every free variable occurring in E with type $(\tau_1 \ldots \tau_p \to \tau_0)$ to the function $\lambda y_1 \ldots y_p.\, w_\tau$.

Consider any flexible-flexible equation

$$e = (\lambda x_1 \ldots x_n.\, y_{(\tau_1 \ldots \tau_p \to \tau_0)}\ u_1 \ldots u_p \approx \lambda x_1 \ldots x_n.\, z_{(\tau'_1 \ldots \tau'_q \to \tau_0)}\ v_1 \ldots v_q).$$

Then $e\theta$ equals $\lambda x_1 \ldots x_n.\, w_\tau \approx \lambda x_1 \ldots x_n.\, w_\tau$.

We say that a unification problem with only flexible-flexible equations is a *solved* unification problem.

The Whole Procedure

A reasonable strategy consists in applying Fail and Simplify eagerly, and Generate only when there is no rigid-rigid equation left.

Generate is non-deterministic, making this procedure branching.

Theorem 4.2 *The procedure made of the rules Fail, Simplify, and Generate is sound and complete.*

Note that the procedure does not enumerate a complete set of unifiers, if there are flexible-flexible pairs left at the end.

Higher-order unification is only semi-decidable.

When solutions exist, Huet's algorithm will find one and terminate, but when there is no solution, it may loop forever.

4.5 Resolution in Higher-Order Logic

To avoid enumerating infinitely many unifiers, Huet's HO resolution procedure delays the computation of unifiers by using constraints.

Once a contradiction has been derived, the corresponding unification problem can then be solved using Huet's pre-unification algorithm.

Resolution:
$$\frac{D \vee B \; [\![K_1]\!] \qquad C \vee \neg A \; [\![K_2]\!]}{D \vee C \; [\![K_1 \wedge K_2 \wedge A \approx B]\!]}$$

Factoring:
$$\frac{C \vee A \vee B \; [\![K]\!]}{C \vee A \; [\![K \wedge A \approx B]\!]}$$

The resolution rules are not always sufficient to guess the necessary substitution based on the available terms.

Example 4.3 *Consider the formula $x_{(o)}$ where x is a Boolean variable. The set $\{x\}$ is saturated by resolution, but still the formula x is unsatisfiable. However, we can guess the substitution $\sigma = \{x \mapsto \neg y\}$ and resolution can then derive the empty clause from x and $\neg y$.*

To overcome this issue, Huet introduces additional *splitting rules*, e. g.,

$$\frac{C \vee A \; [\![X]\!]}{C \vee \neg x_{(o)} \; [\![X \wedge A \approx \neg x]\!]}$$

$$\frac{C \vee A \; [\![X]\!]}{C \vee x_{(o)} \vee y_{(o)} \; [\![X \wedge A \approx (x \vee y)]\!]}$$

With Huet's set of splitting rules, resolution is sound and refutationally complete.

In practice, several improvements are possible.

As soon as a constraint becomes unsatisfiable, delete the corresponding clause.

If a constraint has a small enough set of solutions, generate all applied clauses to replace the constrained original one.

4.6 Superposition in Higher-Order Logic

HO resolution lacks built-in equality, ordering restrictions, and a redundancy concept and is therefore very inefficient in practice.

Can we extend superposition to HO logic?

λ-Free Higher-Order Logic

Step 1: λ-free higher-order logic:

higher-order functions,

partially applied functions,

variables ranging over function types,

but no λ-abstraction

Can in principle be encoded in FOL using an "apply" function:

$f \; (g \; y) \; (x \; g) \;\; \rightarrow \;\; \mathrm{app}(\mathrm{app}(f, \mathrm{app}(g, y)), (\mathrm{app}(x, g)))$,

but encoding is inconvenient for superposition provers.

Extending the superposition calculus to this scenario is rather straightforward, except for one problem:

Reduction orderings are compatible with contexts:
$t > t' \;\Rightarrow\; s \; t > s \; t'$

but typically not with arguments:
$s > s' \;\not\Rightarrow\; s \; t > s' \; t.$

Solution:

Restrict superpositions to "green subterms", that is, subterms that correspond to first-order subterms.

To cover other replacements, add an inference rule

$$\frac{C \vee s \approx s'}{C \vee s\ x \approx s'\ x} \qquad \text{(ArgCong)}$$

Consider applied and non-applied occurrences of the same symbol as distinct when checking redundancy.

Additionally: more complicated variable condition.

Higher-Order Logic Without First-Class Booleans

Step 2: Higher-order logic without first-class Booleans:

+ λ-abstractions,

but Boolean terms only at the outermost level.

New problems:

There are no ground-total simplification ordering upto β-conversion.

Unification is infinitary.

Subterms of ground instances do no longer correspond to subterms of non-ground clauses.

Solution:

Calculus works on $\beta\eta$-short terms; ordering restrictions are sometimes relaxed.

Enumerate CSU using a full unification procedure (e. g., Jensen and Pietrzykowski, Snyder and Gallier, or Vukmirović, Bentkamp, and Nummelin), use dovetailing.

New inference rule

$$\frac{D \vee t \approx t' \qquad C \vee s[u] \approx s'}{(D \vee C \vee s[z\ t'] \approx s')\sigma} \qquad \text{(FluidSup)}$$

if $\sigma \in \mathrm{CSU}(z\ t, u)$.

Full Higher-Order Logic

Step 3: Full higher-order logic:

No more restrictions for occurrences of Boolean terms and variables.

Solution:

HO calculus is based on non-clausal FO superposition.

Hoisting rules, e. g.,

$$\frac{C[t_{(o)}]}{C[\bot] \vee t \approx \top} \qquad \frac{C[t_{(o)}]}{C[\top] \vee t \approx \bot}$$

+ Fluid hoisting rules

+ Rules to "invent" Boolean formula structure.

Developed and implemented in Zipperposition (Bentkamp et al.)

Successful in practice, in particular for problems coming from interactive proof assistents (mainly FO, but with some HO part).

Alternative Approach: Combinators

All λ-terms can be constructed by composing the following *combinators*:

$S = \lambda x\, y\, z.\, x\ z\ (y\ z)$

$K = \lambda x\, y.\, x$

$I = \lambda x.\, x$

Instead of adding λ-abstractions, one can add S, K, and I to the λ-free calculus.

Problem: Choose a good ordering.

Literature

Peter B. Andrews: An introduction to mathematical logic and type theory - to truth through proof. Computer science and applied mathematics, Academic Press, ISBN 978-0-12-058535-9, pp. I-XV, 1-304, 1986.

Peter B. Andrews: Classical Type Theory. Handbook of Automated Reasoning: 965-1007, 2001.

Heiko Becker, Jasmin Christian Blanchette, Uwe Waldmann, Daniel Wand: A Transfinite Knuth-Bendix Order for Lambda-Free Higher-Order Terms. CADE: 432-453, 2017.

Alexander Bentkamp, Jasmin Blanchette, Simon Cruanes, Uwe Waldmann: Superposition for Lambda-Free Higher-Order Logic. Log. Meth. Comput. Sci., 17(2):1:1-1:38", 2021.

Alexander Bentkamp, Jasmin Blanchette, Sophie Tourret, Petar Vukmirović, Uwe Waldmann: Superposition with Lambdas. J. Autom. Reason, 65(7):893–940, 2021.

Alexander Bentkamp, Jasmin Blanchette, Sophie Tourret, Petar Vukmirović: Superposition for Full Higher-Order Logic. CADE-28, 396–412, 2021.

Christoph Benzmüller, Dale Miller: Automation of Higher-Order Logic. Computational Logic: 215-254, 2014.

Jasmin Christian Blanchette, Uwe Waldmann, Daniel Wand: A Lambda-Free Higher-Order Recursive Path Order. FoSSaCS: 461-479, 2017.

Alonzo Church: A formulation of the simple theory of types. Journal of Symbolic Logic, 5(2):56-68, 1940.

Gilles Dowek: Higher-Order Unification and Matching. Handbook of Automated Reasoning: 1009-1062, 2001.

Melvin Fitting: Types Tableaus and Gödel's God. Studia Logica 81(3): 425-427, 2005.

Gérard P. Huet: A Mechanization of Type Theory. IJCAI: 139-146, 1973.

Petar Vukmirović, Alexander Bentkamp, Visa Nummelin: Efficient Full Higher-Order Unification, FSCD 2020, 5:1–5:17, 2020.

Petar Vukmirović, Alexander Bentkamp, Jasmin Blanchette, Simon Cruanes, Visa Nummelin, Sophie Tourret: Making higher-order superposition work, CADE-28, 415–432, 2021.