New Topic!

Higher-Order Logic

Introduction

- **Specifications in first order logic can be clumsy**
- **Higher-Order Logic is widely used since**
  - **it is more expressive than first-order logic**
  - **specifications are often more natural**
  - **they are less likely to contain mistakes and are easier to understand**
- **The main differences between first and higher-order logic are**
  - **no distinction is made between terms and statements**
  - **functions and predicates are treated as first class objects**
- **Higher-order logic is a foundational system like set theory**
  - **can develop mathematics inside higher-order logic**

Overview

- **Basic notation**
- **Types**
- **Special syntax to make notation user-friendly**
- **Definitions**

Higher-Order Logic

- **Familiar notation:**

Term	*Meaning*
$P(x)$	x **has property** P
$\neg t$	**not** t
$t_1 \ \wedge \ t_2$	t_1 **and** t_2
$t_1 \ \vee \ t_2$	t_1 **or** t_2
$t_1 \Rightarrow t_2$	t_1 **implies** t_2
$\forall x.\ t[x]$	**for all** x **it is the case that** $t[x]$
$\exists x.\ t[x]$	**for some** x **it is the case that** $t[x]$

- **Higher-order logic is also called:**
  - *higher-order predicate calculus*
  - *simple type theory*

Formulae Become Terms

- **In higher-order logic statements (formulae) are boolean terms**
  - **i.e. terms whose value is one of the two truth values (or booleans) `T` or `F`**

- **There are only four kinds of primitive terms in higher-order logic**
  - **Variables** x, y, P **etc**
  - **Constants stand for fixed values (e.g. numerals** 0, 1 **etc)**
  - **Function applications have the general form** $t_1\ t_2$ **where** t_1 **and** t_2 **are terms, an example is** $P\ 0$
  - **Lambda-terms denote functions and have the form** $\lambda x.\ t$ **(where** x **is a variable and** t **a term)**

- **Notation on previous slide is 'syntactic sugar' (explained later)**

Statements

- **Statements are just terms in higher-order logic**
  - $\neg t$ is the application of constant $\neg$ to term t
  - $t_1 \Rightarrow t_2$ is the infixed application of constant $\Rightarrow$ to terms t_1 and t_2
- **Only a single variable binding mechanism: λ-abstraction**
  - quantifiers $\forall$ and $\exists$ are regarded as constants
  - quantification syntax $\forall x.\ t$ and $\exists x.\ t$ abbreviates $\forall(\lambda x.\ t)$ and $\exists(\lambda x.\ t)$
- **Example: the term**

  $$\forall n.\ P(n) \Rightarrow P(n+1)$$

  is written instead of

  $$\forall(\lambda n.\ \Rightarrow(P(n))(P(+\ n\ 1)))$$

Higher-Order Variables

- **Higher-order logic generalises first order logic by allowing higher-order variables**
  - **i.e. variables ranging over functions and predicates**

- **The induction axiom for natural numbers can be written as:**

$$\forall P.\ P(0) \wedge (\forall n.\ P(n) \Rightarrow P(n+1)) \Rightarrow \forall n.\ P(n)$$

- **The legitimacy of simple recursive definitions can be expressed by:**

$$\forall n_0.\ \forall f.\ \exists s.\ (s(0) = n_0) \wedge (\forall n.\ s(n+1) = f(s(n)))$$

- **Sentences like these are not allowed in first order logic**
  - **in the first example above P ranges over predicates**
  - **in the second example f and s range over functions**

Floyd-Hoare Logic

- **Correctness specifications can be expressed in higher-order logic**

- **A predicate $\texttt{Spec}$ is defined by**

$$\texttt{Spec}(p, c, q) \ = \ \forall s_1\ s_2.\ p\ s_1\ \wedge\ c(s_1, s_2) \Rightarrow q\ s_2$$

- **$\texttt{Spec}$ is a predicate on triples (p, c, q) where p and q are unary predicates and c is a binary predicate**

- **$\texttt{Spec}$ can't be defined in first order logic**

Constants

Examples of constants are:

- **the equals sign ($=$), the equivalence symbol ($\equiv$),**
- **the negation symbol ($\neg$),**
- **the conjunction symbol ($\wedge$), the disjunction symbol ($\vee$), the implication symbol ($\Rightarrow$),**
- **the universal quantifier ($\forall$), the existential quantifier ($\exists$),**
- **the pairing symbol (comma: $,$),**
- **the numerals 0, 1, 2 etc.,**
- **the arithmetic functions $+$, $-$, $\times$ and $/$, and**
- **the arithmetic relations $<$, $>$, $\leq$ and $\geq$.**

Function applications

- Terms of the form $t_1\ t_2$ are called **applications** or *combinations*
  - can write $f(x)$ instead of $f\ x$
- The subterm t_1 is called the **operator** (or **rator**)
- The term t_2 is called the **operand** (or **rand** or **argument**)
- The result of such a function application can itself be a function and thus terms like $(t_1(t_2))(t_3)$ are allowed
- Functions that take functions as arguments or return functions as results are called **higher-order**
- Function applications can be written as $f\ x$ instead of $f(x)$
  - $t_1\ t_2\ t_3\ \cdots\ t_n$ abbreviates $(\ \cdots\ ((t_1\ t_2)\ t_3)\ \cdots\ t_n)$
  - i.e. application associates to the left

Lambda-terms

- **An abstraction is a term of the form $\lambda x.\ t$ (where t is a term)**

- **$\lambda x.\ t$ denotes the function f defined by:**

$$f(x) \ = \ t$$

- **Example:** **$\lambda n.\ \texttt{factorial}(\texttt{square}(n))$ denotes function f where:**

$$f(n) \ = \ \texttt{factorial}(\texttt{square}(n))$$

- **The variable x and term t are called respectively the bound variable and body of the λ-expression $\lambda x.\ t$**

Types

- **Consider the predicate $\mathtt{P}$ defined by:**

$$\mathtt{P}\ x \;=\; \neg(x\ x)$$

- **From this definition it follows that:**

$$\mathtt{P}\ \mathtt{P} \;=\; \neg(\mathtt{P}\ \mathtt{P})$$

- **This is a version of Russell's paradox**

- **Russell invented types to prevent such inconsistencies**

The Structure of Types

- **Types denote sets of values: they are atomic or compound**
- **Two kinds of atomic types:**
  - type constants, e.g. num or $bool$
  - type variables: α, β, γ etc
- **Compound types are built using type operators**
  - $\rightarrow$ and $\times$ are infixed binary type operators
- **If σ, σ_1 and σ_2 are types then so are $\sigma_1 \rightarrow \sigma_2$ and $\sigma_1 \times \sigma_2$**
  - $\sigma_1 \rightarrow \sigma_2$ denotes the set of functions with domain σ_1 and range σ_2
  - $f : \sigma_1 \rightarrow \sigma_2$ means f a function from σ_1 to σ_2
  - $\sigma_1 \times \sigma_2$ denotes the Cartesian product type
  - $p : \sigma_1 \times \sigma_2$ means p a pair (x, y) where $x : \sigma_1$ and $y : \sigma_2$

Well-Typed Terms

- Terms must be **well-typed**
- Typing rules are essentially those of ML
- Must be possible to assign a type to each subterm so 1 and 2 hold:
  1. For every subterm $t_1\ t_2$ there are types σ and σ' such that:
     (a) t_1 is assigned $\sigma' \rightarrow \sigma$
     (b) t_2 is assigned σ'
     (c) $t_1\ t_2$ is assigned the type σ.
  2. Every subterm of the form $\lambda x.\ t$ is assigned a type $\sigma_1 \rightarrow \sigma_2$ where:
     (a) x is assigned σ_1
     (b) t is assigned σ_2.

Notation

- **Writing $t{:}\sigma$ indicates that a term t has type σ**

- $\sigma_1{\to}\sigma_2{\to}\sigma_3{\to}\ \cdots\ \sigma_n{\to}\sigma$ **abbreviates** $\sigma_1{\to}(\sigma_2{\to}(\sigma_3{\to}\ \cdots\ (\sigma_n{\to}\sigma)\ \cdots\ ))$
  - **i.e. $\to$ associates to the right**

- $\lambda x_1\ x_2\ \cdots\ x_n.\ t$ **abbreviates** $\lambda x_1.\ \lambda x_2.\ \ \cdots\ \lambda x_n.\ t$

- **The scope of the "." after a λ extends as far to the right as possible**
  - **for example,** $\lambda b.\ b = \lambda x.\ \mathtt{T}$ **means** $\lambda b.\ (b = (\lambda x.\ \mathtt{T}))$ **not** $(\lambda b.\ b) = (\lambda x.\ \mathtt{T})$

Types of Constants

- **All constants are assumed to have a fixed type**
- **If this type is polymorphic (i.e. contains type variables) then for the purposes of type checking the constant behaves as though it is assigned every instance of the type**
- **For example, if `twice` had type $(\alpha\to\alpha)\to(\alpha\to\alpha)$, then the terms `twice`$($`sqrt`$)$ and `twice`$($`not`$)$ would be well-typed**

Special Syntactic Forms

- **Certain applications are conventionally written in special ways, for example:**
  - $+\ t_1\ t_2$ **is written** $t_1 + t_2$
  - $,\ t_1\ t_2$ **is written** (t_1, t_2)
  - $\forall(\lambda x.\ t)$ **is written** $\forall x.\ t$
- $+$ **and** $,$ **are examples of** **infixes**
- $\forall$ **is an example of a** **binder**
- **Some other** *ad hoc* **syntactic forms are also used (e.g. conditionals)**

Infixes

- **If `f` is an infixed constant then applications are written as t_1 `f` t_2 rather than as `f` t_1 t_2**

- **Standard examples of infixes are the arithmetic functions $+$, $\times$ etc.**

- **Whether a constant is an infix or not has no logical significance, it is merely syntactic**

- **Examples of infixes are the following constants:**

  $\wedge :\ bool{\rightarrow}bool{\rightarrow}bool$ **(Conjunction)**
  $\vee :\ bool{\rightarrow}bool{\rightarrow}bool$ **(Disjunction)**
  $\Rightarrow:\ bool{\rightarrow}bool{\rightarrow}bool$ **(Implication)**
  $\equiv:\ bool{\rightarrow}bool{\rightarrow}bool$ **(Equivalence)**

  Equality is also an infixed constant; it is polymorphic:

  $=:\ \alpha{\rightarrow}\alpha{\rightarrow}bool$

  Equivalence ($\equiv$) is equality ($=$) restricted to booleans

Binders

- It is sometimes more readable to write $\mathtt{f}\ x.\ t$ instead of $\mathtt{f}(\lambda x.\ t)$
- For example, the quantifiers $\forall$ and $\exists$ are polymorphic constants:
  $\forall :\ (\alpha{\rightarrow}bool){\rightarrow}bool$
  $\exists :\ (\alpha{\rightarrow}bool){\rightarrow}bool$
- The idea is that if $P : \sigma{\rightarrow}bool$, then $\forall(P)$ is true if $P(x)$ is true for all x and $\exists(P)$ is true if $P(x)$ is true for some x
- Instead of writing $\forall(\lambda x.\ t)$ and $\exists(\lambda x.\ t)$ it is nice to be able to use the more conventional forms $\forall x.\ t$ and $\exists x.\ t$
- **Example:** recall the statement of mathematical induction:

$$\forall P.\ P(0) \wedge (\forall n.\ P(n) \Rightarrow P(n+1)) \Rightarrow \forall n.\ P(n)$$

  This is a term of type $bool$

- Without infix and binder notation terms are less readable

$$\forall(\lambda P.\ {\Rightarrow}(\wedge(P\ 0)(\forall(\lambda n.\ {\Rightarrow}(P\ n)(P(+\ n\ 1)))))(\forall(\lambda n.\ Pn)))$$

Pairs and Tuples

- **A function of n arguments can be represented as a higher-order function of 1 argument that returns a function of $n-1$ arguments**

- **Thus $\lambda m.\ \lambda n.\ m^2+n^2$ represents the 2 argument function that sums the squares of its arguments**

- **Functions of this form are called curried**

- **An alternative way of representing multiple argument functions is as single argument functions taking tuples as arguments**

- **If $t_1{:}\sigma_1$ and $t_2{:}\sigma_2$ then (t_1, t_2) has type $\sigma_1{\times}\sigma_2$**
  - (t_1, t_2) **denotes a pair**

Paired Functions

- **Another representation of the sum-squares function would be as a constant, `sumsq` say, of type $(num \times num) \to num$ defined by:**

$$\texttt{sumsq}(m, n) \;=\; m^2 + n^2$$

- **A term of the form (t_1, t_2) is equivalent to the term $,\ t_1\ t_2$ where "$,$" is a polymorphic infixed constant of type $\alpha \to \beta \to (\alpha \times \beta)$**

- **Instead of having tuples as primitive, they will be treated as iterated pairs. Thus the term:**

$$(t_1, t_2,\ \ldots\ , t_{n-1}, t_n)$$

  is an abbreviation for:

$$(t_1, (t_2,\ \ldots\ , (t_{n-1}, t_n)\ \ldots\ ))$$

  i.e. "$,$" associates to the right

- **To match this, the infixed type operator $\times$ also associates to the right so that if $t_1{:}\sigma_1$, $\ldots$, $t_n{:}\sigma_n$ then:**

$$(t_1,\ \ldots\ , t_n) : \sigma_1 \times\ \cdots\ \times \sigma_n$$

Definitions

- **Definitions have the form** $\vdash\ \mathsf{c} = t$ **where** c **is a new constant and** t **is a closed term that doesn't contain** c

- **Such a definition is a particular kind of axiom that introduces the constant** c **as an abbreviation for the term** t

- **The requirement that** c **may not occur in** t **prevents definitions from being recursive, this is to rule out inconsistent 'definitions' like**

$$\vdash\ \mathsf{c} = \mathsf{c} + 1$$

- **Example: a constant** `Cond` **can be defined (see the notes) so that** `Cond` $t\ t_1\ t_2$ **means "if** t **then** t_1 **else** t_2**"**
  - **the special syntax** $(t \to t_1 \mid t_2)$ **is used for such terms**

Function Definitions

- **A function definition:**

$$\vdash \mathtt{f} \ = \ \lambda x_1 \ \cdots \ x_n.\ t$$

can be written as:

$$\vdash \ \mathtt{f}\ x_1 \ \cdots \ x_n \ = \ t$$

- **Definitions have the property that adding a new definition to the set of existing ones cannot introduce any inconsistencies**

- **As was shown by Russell and Whitehead, with suitable definitions, all of classical mathematics can be constructed from logic**

- **See notes for how recursive definitions are handled**

Type Definitions

- New types can be defined
- For example, can define a new type `three` with values `ONE`, `TWO` and `THREE`
- Can also add recursive type definitions
- For example, could define a new type α `list` with constructors `NIL` and `CONS`
- Such definitions correspond closely to datatype definitions in ML
- Recursion theorems that allow ML style recursive definitions over such recursive types can then be proved
- Details of type definitions not given here

Summary

- **Primitive notions**
  - **variables, constants, function applications, λ-abstractions**
- **Types**
  - **statements (formulae) are just boolean terms**
- **Special syntax to make notation user-friendly**
  - **infixes, binders, conditional notation**
- **Definitions**
  - **introduces new constants consistently**