

Interactive Theorem Proving in Higher-Order Logic

First-Order logic vs. Higher-Order Logic

Restrictions of first-order logic:

- Quantification only over object variables, not over function or predicate variables.
- Description of arithmetic not possible.

Problems with higher-order logic:

- Russell paradoxes
- Gödel's incompleteness theorem

Russell Paradoxes and Types

Higher-order quantification means that it should be possible to quantify over predicates, e.g. in induction:

$$\forall P. (P(0) \wedge (\forall n. P(n) \Rightarrow P(s(n)))) \Rightarrow \forall n. P(n)$$

The problem is with the "definition" of a predicate, P , as $\forall x. P(x) \Leftrightarrow \neg x(x)$. This results in $P(P) \Leftrightarrow \neg P(P)$ - i.e. an absurdity.

Russell's solution: disallow expressions of the kind $P(P)$ by introducing types. Every variable and constant gets a type, either a base type or a function type, compound terms must respect type restrictions. In particular, $P(P)$, cannot be well-typed.

Semantics of Higher-Order Logic

Map terms of type τ into a non-empty universe \mathcal{D}_τ .

Standard Semantics The universe of function (and predicate) types is the set of all functions from the domains of the argument types to the domain of the result type. This is incomplete.

"Henkin" Semantics The universe of function types is the set of all functions *expressible in the language* from the relevant domains etc. This is complete although it does map onto a (clumsy and unusable) version of first-order logic.

The Logic of Computable Functions

- Milner's name for a logic devised by Dana Scott in 1969 but not published until 1993.
- Terms are from the typed λ -calculus and formulae are from predicate logic.
- Types are interpreted as Scott domains (CPOs).
- The logic is intended for reasoning, using fixed-point induction, about recursively defined functions of the sort used in denotational semantic definitions.

The "LCF Paradigm" (Milner)

Begins with "Stanford LCF" and has a number of successors including "Edinburgh LCF", "Cambridge LCF" and many current day interactive theorem proving assistants including: Isabelle (Paulson), HOL (Gordon), Nuprl (Constable) and Coq (Huet). Many of these systems no longer use Scott's logic. The basics of Stanford LCF were as follows

- Declare a main goal – a formula in Scott's logic.
- Split the goal into subgoals – using a fixed set of subgoaling commands.
- Subgoals are then either solved using a simplifier or split into more subgoals.
- These commands create data structures representing a formal proof, which is the output of the proof process.

Edinburgh LCF

- Solved the problem of a fixed set of subgoaling commands by inventing a metalanguage (ML) for scripting proof commands.
- Key idea: Have a type called `thm`. The only values of type `thm` are axioms or are obtained from axioms by applying the inference rules.
- Language requirements: abstract type for encapsulating theorems, strict typechecking so theorems can only be created by inference rules, support for composing proof scripts.
- Milner invents the idea of *tactics* (and *tacticals*) for securely programming proof scripts.
 - A tactic is a function from goals to subgoals but it is also more than that. It provides a way of constructing a proof data structure.
 - Tacticals are higher-order functions for composing tactics.

Forward versus Backward Proof

- Forward proof consists of applying inference rules to axioms or previously proved theorems.
 - Stanford LCF works backwards from the hypothesis (or goal) to be proved until the axioms are reached.
1. $(\phi \Rightarrow (\psi \Rightarrow \theta)), (\phi \Rightarrow \psi), \phi \vdash \phi$. Assumption.
 2. $(\phi \Rightarrow (\psi \Rightarrow \theta)), (\phi \Rightarrow \psi), \phi \vdash \phi \Rightarrow (\psi \Rightarrow \theta)$. Assumption.
 3. $\Gamma \vdash (\psi \Rightarrow \theta)$. Modus Ponens.
 4. $\Gamma \vdash (\phi \Rightarrow \psi)$. Assumption.
 5. $\Gamma \vdash \psi$. Modus Ponens.
 6. $\Gamma \vdash \theta$. Modus Ponens.
 7. $(\phi \Rightarrow (\psi \Rightarrow \theta)), (\phi \Rightarrow \psi) \vdash \phi \Rightarrow \theta$. \Rightarrow Introduction.
 8. $(\phi \Rightarrow (\psi \Rightarrow \theta)) \vdash (\phi \Rightarrow \psi) \Rightarrow (\phi \Rightarrow \theta)$. \Rightarrow Introduction.
 9. $\vdash (\phi \Rightarrow (\psi \Rightarrow \theta)) \Rightarrow ((\phi \Rightarrow \psi) \Rightarrow (\phi \Rightarrow \theta))$. \Rightarrow Introduction.

The Logicians Notion of Proof

- Classical (Hilbert) notion of formal proof: a proof is a sequence of lines such that each line
 - is an axiom
 - follows from preceding lines by a rule of inference.The last line is the theorem proved.
- Thus formal logical proof is forward.
- Milner's idea was to use goal oriented proof search to generate formal forwards proofs.

Tactics

- A tactic, `T`, is a function:
`T: goal -> goal list x (thm list -> thm)`
- Suppose that for a given goal $g - T(g) = ([g1, \dots, gN], f)$
 - If theorems `th1, \dots, thN` solve the goals `g1, \dots, gN`
 - then $f([th1, \dots, thN])$ should solve `g`.
- When `T` has this property it is called *valid*.
- Invalid tactics don't prove invalid theorems, they just generate unhelpful (i.e. unsolvable) subgoals.

Basic Tactics and Tacticals

- Basic tactics are “inverted inference rules”

$$\frac{\vdash A \quad \vdash B}{\vdash A \wedge B} \wedge\text{-Introduction}$$

The tactic splits the goal $\vdash A \wedge B$ into the two subgoals A and B and the function is $\lambda x. \lambda y. \wedge - I(x, y)$

- Tacticals let you compose these into more complex tactics such as $\wedge - I$ THEN $\vee - I$ or REPEAT $\wedge - I$. So that a user can take large steps as they direct the proof.

Advanced Tactics

“Big” proof steps are not necessarily programmed up simply by composing tactics. For instance many interactive theorem provers contain resolution proof tactics for automatically finding proofs in first-order logic. These tactics contain an (efficient) resolution prover written in ML and the proof trace from this is then translated into the justification function. They do not work by REPEAT RESOLVE.

This is why you will occasionally successfully apply a tactic only to find that the proof at the end can not be generated.

Theories

- Most people working with an interactive theorem prover want to use a set of definitions of their own.
- Generally this involves developing *theories*. Sets of definitions, theorems and proof support tools that live together.
- Most interactive theorem provers have increasingly rich libraries of such theories.

Deep and Shallow Embeddings

- Users often want to formalise some other language: its syntax and semantics within the language of the interactive theorem prover.
- The can either specify a type for expressions in the new language and then define the syntax and semantics for this type. This is generally quite difficult with the user having to prove well-formedness properties and develop their own proof support. However it is the best way to proceed if you are interested in reasoning about the new language. This is called a *deep* embedding.
- On the other hand you can reuse much of the existing material. For instance you can model parts of the new language as natural numbers, or lists or other types existing in the theorem prover. You then inherit well-formedness from these types and all the proof support developed for them. It is impossible to reason about another language if you do this but it is generally easier to reason with the new language. This is called a *shallow* embedding.

Definitions

- Introducing new concepts and inference rules as *definitions* is considered a safe way of extending your trusted core.
- If $M : \sigma$ has no free variables, M does not contain the identifier c , there are no type variables in M not also present in σ and c is not already the name of a constant, then a new constant c with generic type σ can be defined by extending the syntax of the logic to include c as a constant and adding the axiom $\vdash c = M$.
- If $P : \sigma \rightarrow bool$ has no free variables, both σ and p contains no type variables, $\vdash \exists x. P(x)$ if a theorem of the logic ...
- For recursive definitions you must be able to prove $\vdash \exists x. x = M[x]$.

Most theorem provers provide a lot of support for definitions e.g. by exploiting the primitive recursion theorem

$$\vdash \forall x. f. \exists fn : num \rightarrow \alpha. (fn(0) = x) \wedge (\forall n. fn(Suc(n)) = f(fn(n), n))$$

Isabelle: A Generic Theorem Prover

- Most LCF theorem provers have some logic as their trusted core. Therefore they can only be used for proofs in that Logic.
- Isabelle aims to be a *generic* theorem prover. Inference rules are represented as generalised Horn Clauses and are applied using resolution. However the term language is that of Higher-Order Logic.
- Horn clauses are clauses which have only one positive literal i.e. they are of the form: $P \vee \neg Q_1 \vee \dots \vee \neg Q_n$ or $Q_1 \wedge \dots \wedge Q_n \rightarrow P$.

- For instance

$$\frac{P \quad Q}{P \wedge Q}$$

becomes

$$P, Q \Rightarrow P \wedge Q$$

- \wedge is a constant in the *object logic* while \Rightarrow is "and" in the *meta-logic*.

- Another Example:

$$\frac{\begin{array}{cc} [P] & [Q] \\ P \vee Q & \vdots \\ & \vdots \\ & R \quad R \end{array}}{R} \\ [P \vee Q, P \Rightarrow R, Q \Rightarrow R] \Rightarrow R$$

- Since the terms may be higher order and may contain function variables and λ -abstractions you have to use Higher Order Unification.
- Isabelle is a *logical framework*.

Further Reading

HOL <http://www.cl.cam.ac.uk/Research/HVG/HOL/>
Isabelle <http://www.cl.cam.ac.uk/Research/HVG/Isabelle/>
PVS <http://pvs.csl.sri.com/>
Coq <http://pauillac.inria.fr/coq/coq-eng.html>
Nuprl <http://www.cs.cornell.edu/Info/Projects/NuPrl/>
LEGO <http://www.dcs.ed.ac.uk/home/lego/>