MAU11602
Lecture 17
2026-02-26

Existential presuppositions

Suppose we want to apply first order logic with the integers as our domain. We have two choices: make sure all functions are total or use a version of FOL where not every expression of integer type necessarily designates an actual integer.

Functions can fail to be total in a predictable way, e.g. division by zero, or an unpredictable way, e.g. non-termination of a recursively defined function.

Actually there's a third option: pretend that you're only going to use total functions, and so can use a simpler version of FOL, but then cheat and start using partial functions without modifying the rules of inference.

This is unsound, but is generally what people do.

In practice it leads to trouble less often than you'd expect, because mathematicians develop an instinct for when a supposedly safe rule of inference is not really safe.

I am not a fan of this approach.

Example

Hopefully you already know that 1 is the smallest positive integer, but did you know it's also the smallest positive real number? Here's a proof.

I will use the following facts from real analysis.

- For all real x we have $x > 0$, $x = 0$, or $x < 0$, i.e. $\forall x.(x > 0 \vee x = 0 \vee x < 0)$.
- If I is an open interval and $a_0, a_1, \ldots$ is a sequence with $\lim_{n\to\infty} a_n \in I$ then $a_n \in I$ for all but finitely many values of n.

Consider the sequence $a_n = (-1)^n$. By the first fact we have $\lim_{n\to\infty} a_n > 0$, $\lim_{n\to\infty} a_n = 0$, or $\lim_{n\to\infty} a_n < 0$.

Applying the second fact to $I = (0, \infty)$ rules out the first possibility, and applying it to $I = (-\infty, 0)$ rules out the third, so $\lim_{n\to\infty} a_n = 0$.

Suppose there were positive real number smaller than 1. Call it ϵ and apply the second fact to $I = (-\epsilon, \epsilon)$. This gives a contradiction, so there is no such number.

In fact an intuitionist wouldn't accept either the first fact or the proof by contradiction, but the real problem with this proof is the substitution of $\lim_{n\to\infty}(-1)^n$ for x in the statement $\forall x.(x > 0 \vee x = 0 \vee x < 0)$.

FOL without existential presuppositions

The problem with the preceding proof comes from the fact that $\lim_{n\to\infty}(-1)^n$ does not exist, but our rule of inference for $\forall$ elimination assumes that every expression of the correct type represents an actual element of the domain.

We need a way to say that an object represented by an expression does in fact exist, and we need rules of inference which reflect this.

The statement $\lim_{n\to\infty} a_n$ exists should be written as $\exists x.x = \lim_{n\to\infty} a_n$. More generally, $\exists X.X = T$ means the expression T refers to an actual individual in our domain.

We also need to change our rules of inference:

$$\frac{\Gamma \vdash P[A/X]}{\Gamma \vdash \forall X.P} \qquad \frac{\Gamma\,,\,(\exists X.X = T) \wedge P[T/X] \vdash Q}{\Gamma\,,\,\forall X.P \vdash Q}$$

$$\frac{\Gamma \vdash (\exists X.X = T) \wedge P[T/X]}{\Gamma \vdash \exists X.P} \qquad \frac{\Gamma, P[A/X] \vdash Q}{\Gamma, \exists X.P \vdash Q}$$

Informal reasoning

The main point of formalising mathematical arguments is to learn not to have to formalise them.

If you understand how to formalise arguments well enough you reason informally while avoiding the errors, e.g. variable capture, that formalisation is meant to prevent.

You can also learn this from experience, just as it's possible to learn to speak a language correctly without systematically studying its grammar.

That can be slow and painful though. Also, it makes it hard to explain why something is wrong, even if you recognise that it is.

Mathematicians mostly learn to apply logic correctly, without being able to articulate it. Students often make the kind of mistake we saw in the proof that 1 is the smallest positive real, but working mathematicians don't.

Informal mathematical language is often ambiguous though. For example, no one follows FOL's distinction between individual constants and variables, even though they obey different rules of inference. The reader is just meant keep track of how each letter is being used.

Totality, again

In a logic with existential presuppositions every expression represents a total function, so we can't allow things like division, limits, integrals, etc. in our language.
In a logic without existential presuppositions we are allowed to have expressions corresponding to partial functions, but total functions, like addition and multiplication, are still total. How do we express this?

$$\forall x.\forall y.\exists z.z = x + y, \qquad \forall x.\forall y.\exists z.z = x \cdot y.$$

For division we need to make explicit the conditions for the quotient to exist,

$$\forall x.\forall y.(y = 0) \vee \exists z.z = x/y.$$

but once we do then we can safely reason with it.

Computational aspects of partial functions

We finally know how to deal with partial functions and unevaluable expressions in theory, but how do we deal with them in practice?
What should `factorial` do when given a negative argument?
There are a number of choices we could make:
- Return junk: For factorial we could return zero for negative arguments, for example. This makes it the caller's job not to supply negative arguments, which works here because the caller can check. It would work less well, for example, for a table lookup function.
- Use option types, if available: The caller can now check whether the return value is something or nothing.
This only works in a language which has option types or tagged unions.
We can write a function which takes ordinary functions and produces an option version, which checks arguments and proceeds only if they are something, otherwise returning nothing.

Computational aspects of partial functions, continued

- Don't return: We can use non-terminating recursion to make sure the function never returns anything at all.
This is extremely user-hostile, and I've never seen anyone deliberately do it, but I have seen such functions used to reason about the behaviour of other functions.
- Fail immediately: We can just print an error message and stop the program.
This is the standard way to cope with things which should never happen.
In an eager language this can cause a program to fail because of an irrelevant calculation.
- Continuations: Have the caller supply a function to be used in case of failure.
This only works if your language has first class functions, but is a useful trick.
It allows the caller to decide how to handle exceptional cases rather than the callee.
There is a whole programming style based on this: *continuation passing style*. It's tricky to learn, but powerful.

Computational aspects of partial functions, conclusion

- Exceptions: This is really just a prepackaged easier to use form of continuations. Usually functions can raise an exception and can choose to handle certain types of exception. Control passes from the raiser to the handler, which might be the caller, the caller's caller, etc.
If no one is prepared to handle the exception then this reverts to the fail immediately option.
Handling is also called catching. Languages often aren't even internally consist.
- Delimited continuations (resumable exceptions): Normally once an exception is raised there's no way back. If you're willing to use experimental features of niche languages you can allow functions to resume after the cause of the error has been resolved.
Which of these seven approaches is best depends on what your function is meant to do and what your language provides.

Exceptions, continuations, and types

Option types present no theoretical problems for type theory but exceptions and non-termination require some modifications.

Previously every typable expression had a type and a value, and for sound languages the value of the expression has that type.

Some expressions have no type, and that's a feature, not a bug. It prevents us from doing silly things. We never try to assign a value to an untypable expression.

Now we have expressions which have a type, but have no value, either because an expression has a function which raises an expression or doesn't return.

Type safety (soundness) now means that *if* the expression evaluates then its value is of the correct type.

Continuations require even more modifications. In fact it has been proved that the logical counterpart of a language with continuations is classical logic, not minimal or intutionistic logic.

Quantifiers and Curry-Howard

Zeroeth order logic corresponds to a particular version of type theory. What about first order logic?

A *dependent type* is a type which depends on a value, e.g. n-dimensional vectors.

We shouldn't add vectors of different dimensions. A type theory with dependent types can prevent this sort of error.

Dependent types can be very useful, but type inference with dependent types is undecidable.

As you may have guessed, first order logic is the Curry-Howard counterpart of dependent type theory.

First order logic, unlike zeroeth order logic, is undecidable, so dependent type inference, unlike ordinary type inference, is undecidable.

Programming languages with dependent types do exist, but either implement them only partially or are very annoying to use.