Interactive Computer Theorem Proving

Lecture 14: Twelf

CS294-9
November 30, 2006
Adam Chlipala
UC Berkeley

“Proof Assistants”

Proof Assistants

“I'm going to help you construct formal proofs of all sorts of things.”

Isabelle

Coq

....

PVS

Logical Frameworks

“I'm going to help you **represent** *logics* and *programming languages*.”

LF

....

Twelf: a system for proving meta-theorems about LF formalizations

Comparison

- Specialized to definition of logics, languages, and their metatheories.
- A theorem is proved by a typed functional program.
- A **meta-theorem** is proved by a typed **logic** program.
- Designed with representation of **variable binding, assumption contexts**, etc., in mind.
- Approximately zero proof automation.

- Supports theorem proving in general.
- A theorem is proved by a typed functional program.
- No distinction between theorems and meta-theorems.
- No special support for those concepts.
- As much proof automation as you are willing to code.

What's All This “Meta” Business?

- An **object language** is one that you are formalizing.
  - e.g., first-order logic, lambda calculus
- A **meta language** is what you are using to formalize the object language.
  - e.g., LF, Calculus of Inductive Constructions
- A **meta-theorem** is a theorem about theorems.
  - We think of PL typing, evaluation, etc., derivations as theorems.

LF Style, Part I:

A very syntactic approach to definitions

nat : **type**.

O : nat.

S : nat -> nat.

prop : **type**.

true : prop.

false : prop.

not : prop -> prop.

and : prop -> prop -> prop.

or : prop -> prop -> prop.

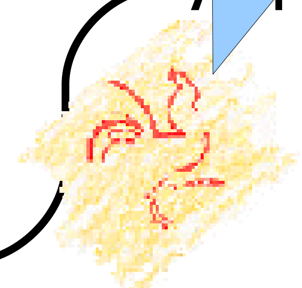
imp : prop -> prop -> prop.

An LF **signature**

The values of each type are precisely those you can build using the specified constant symbols, application, and function abstraction (lambda).

But what about the induction principles you need to reason about these types?

All in good time. :-)



LF/Elf Style, Part II:
Relational definitions of everything

plus : nat -> nat -> nat -> **type**.

plus_O : plus O M M.

plus_Sn : plus N M R

    -> plus (S N) M (S R).

valid : prop -> **type**.

trueI : valid true.

falseE : valid false

    -> valid P.

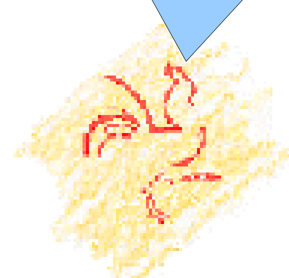
andI : valid P

    -> valid Q

    -> valid (and P Q).

LF doesn't allow recursive function definitions. Rather, we use inference rules to define a relation that happens to be a function.

So derivations of “plus” and “valid” are examples of what we'll be calling “theorems.”



That's right.

LF/Elf Style, Part III:

Meta-theorems as relations over th...

$\forall$ $\exists$

O_id : {*N* : nat} plus *N* O *N* -> **type**.

~~O_id_O~~ : O_id O plus_O.

~~O_id_Sn~~ : O_id *N* *PF*

-> O_id (S *N*) (plus_Sn *PF*).

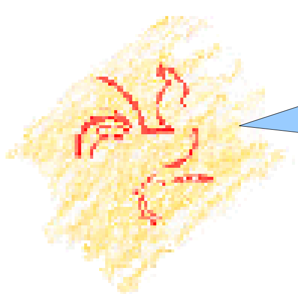
$\forall$ $\exists$

self_imp : {*P* : prop} valid (imp *P* *P*) -> **type**.

self_imp_pf : self_imp *P* (impI ([*H*] *H*)).

You can think of these meta-theorems as **logic programs** for building theorems.

theorem.



OK, let's try running O_id on 1.

Goal: plus (S O) O (S O)

Apply O_id_Sn...

Goal: plus O O O

Apply O_id_O.

Result: O_id_Sn O_id_O

LF/Elf/Twelf Style, Part IV:

Proof checking as totality checking

That seems like a pretty free-form proof format! How do you know a proof is believable?

You've hit on the main issue: Since we follow the proofs-as-programs paradigm, it's critical that we admit only **terminating** programs, be they functional or logic programs.

I have to be on guard for devious "proofs" like this one!

...and I have a bit more work to do than you do to get there, since notions like *functions*, *recursion*, and *case analysis* are **implicit** in my logic programs.

$$(\mathbf{fix}\ \mathrm{F}\ (x : \mathrm{A}) : \mathrm{B} := \mathrm{F}\ x)$$

Totality Checking, Prologue:

Explain input-output behavior

"Interpret this as a logic program computing the outputs from the inputs.
I guarantee that, for any input values with no free variables, any outputs of the logic program will also have no free variables."

This is related to a basic property of Coq functions.

Input **Output**

O_id : {N : nat} plus N O N -> **type**.

O_id_O : O_id O plus_O.

O_id_Sn : O_id N PF

-> O_id (S N) (plus_Sn PF).

Totality Checking, Part I:
Termination

"Every recursive call decreases N,
so any execution terminates."

User-provided argument

This is like the syntactic checks performed on **fix**es.

O_id : {N : nat} plus N O N -> **type**.

O_id_O : O_id O plus_O.

O_id_Sn : O_id N PF

-> O_id (S N) (plus_Sn PF).

Totality Checking, Part II:
Input Coverage

An input of O is matched here...

... and any S input is matched here.

All part of the rules for using **match** in Coq!

O_id : {N : nat} plus N O N -> **type**.

O_id_O : O_id O plus_O.

O_id_Sn : O_id N PF

-> O_id (S N) (plus_Sn PF).

Totality Checking, Part III:
Output Coverage

This variable will match anything returned by the recursive call.

In Coq, you must **match** on recursive results and explicitly handle all cases.

O_id : {*N* : nat} plus *N* O *N* -> **type**.

O_id_O : O_id O plus_O.

O_id_Sn : O_id *N* *PF*

  -> O_id (S *N*) (plus_Sn *PF*).

Interlude

[Demo of natural numbers in Twelf]

LF Style, Part V:
Higher-order abstract syntax

$\forall\, x,\ P(x)$

forall (**fun** x => atomic P x)

In Twelf notation....

forall ([x] atomic P x)

formula : **type**.

symbol : **type**.

object : **type**.

atomic : symbol -> object -> formula.

and : formula -> formula -> formula.

imply : formula -> formula -> formula.

forall : (object -> formula) -> formula.

Higher-order abstract syntax:
Use the meta language binder to encode object language binders.

HOAS for Inference Rules

Finally, we also encode **substitution** using the LF substitution associated with functions.

valid : formula -> **type**.

impI : (valid P -> valid Q) -> valid (imp P Q).

impE : valid (imp P Q) -> valid P -> valid Q.

forallI : ({X} valid (P X)) -> valid (forall P).

forallE : valid (forall P) -> valid (P X).

H : ∀ x, P(x) ⊢ P(x)[v/x]

FOL: **LF:**

encode

encode

H : forall Q[x] atomic P x ⊢ atomic P v

H : valid P ⊢ valid Q

reduce

atomic P v

Negative Occurrences

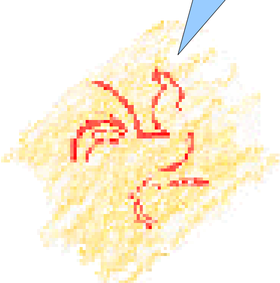
Here's the syntax of untyped lambda calculus. Note that we **don't need a variable case**, because we encode them with LF variables as for “forall” on the last slide!

term : **type**.

app : term -> term -> term.

lam : (term -> term) -> term.

I don't allow inductive type definitions like this.



~~**Inductive** term : **Set** :~~

~~| App : term -> term~~

~~| Lam : (term -> tern~~

Inductive L : **Set** :=

| Lam : (L -> L) -> L.

Defi L :=

No such problems for me because LF is a very restricted language that can't express such things. In particular, there are no case analysis expressions!

Eval com (App delta delta).

Regular Worlds

size : term -> nat -> **type**.

size_app : size *E1 N1*

-> size *E2 N2*

-> plus *N1 N2 N3*

-> size (app *E1 E2*) (S *N3*).

size_lam : ({*X* : term} size *X* O

-> size (*E X*) *N*)

-> size (lam *E*) (S *N*).

The idea of **regular worlds** is used to describe contexts similarly to how regular expressions describe strings. The big difference is that we consider variables up to alpha equivalence in the usual way.

$$\Gamma ::= . \mid \Gamma\ \{X : \mathrm{term}\}\ \{PF : \mathrm{size}\ X\ N\}$$

Canonical Forms

Result #1:

For a fixed LF signature, any term whose type is some constant defined like “$\mathrm{T} : \mathbf{type}$” is equivalent to some application of some constant “$\mathrm{C} : \mathrm{T1}$ -> ... -> Tn -> T”.

Result #2:

For a fixed LF signature, any term of type “$\mathrm{T1}$ -> $\mathrm{T2}$” is equivalent to some “$[x : \mathrm{T1}]\ \mathrm{E}$” where E has type $\mathrm{T2}$.

In Coq, the term might be a $\mathbf{fix}$.

Full Totality Checking

size : term -> nat -> **type**.

size_app : size $E1$ $N1$
-> size $E2$ $N2$
-> plus $N1$ $N2$ $N3$
-> size (app $E1$ $E2$) (S $N3$).

size_lam : ({X : term} size X O
-> size (E X) N)
-> size (lam E) (S N).

({X : term} {PF : size X N})*

The user needs to

Proving termination for size E N

First apply canonical forms to normalize E completely, then induct on the result.

E = app $E1$ $E2$: Only size_app applies, and recursive calls use syntactic subterms.

E = lam E': Only size_lam applies. E' must be some $[Y]\,E''$. $E''\ X$ is as large as E' and strictly smaller than E, so the recursive call is OK.

E is a variable X: The regular world spec guarantees that X has a twinned size hypothesis to use.

Wrap-Up

(before going to code demo)

- **LF**: a language for encoding languages
  - Supports **higher-order abstract syntax**
  - Minimal enough to admit a **canonical forms** property that facilitates simple induction over term structure
- **Twelf**: tool support for checking LF meta-theorems
  - Proof checking as **totality checking** of logic programs