Computational Logic

Interactive Theorem Proving

Week 6

Cezary Kaliszyk (VO)
Vincent van Oostrom (PS)

Nov 11, 2016

Summary

So far

Proof Assistants, HOL Light, $\lambda_{\rightarrow}$, Gentzen-style, Tactics

- Properties of $\lambda_{\rightarrow}$
- BHK interpretation and λ-cube again
- Dependent types
- λ_P

Today

- More involved tactics
- Quotients
- Automated Reasoning

Rules, Conversions, Tactics

- Rule: `thm -> thm`
  - Examples?
- Conversion: `term -> thm`
  - Examples?
- Tactic: goalstate refinement
  - Type?
- Thm_tactical: `thm_tactic -> thm_tactic`
- How to combine them?
  - THEN...
  - DEPTH_CONV
- How to transform one to other?
  - CONV_TAC, CONV_RULE

Rules, Conversions, Tactics

- Rule: `thm -> thm`
  - Examples?
- Conversion: `term -> thm`
  - Examples? `BETA_CONV`
- Tactic: goalstate refinement
  - Type?
- Thm_tactical: `thm_tactic -> thm_tactic`
- How to combine them?
  - THEN...
  - DEPTH_CONV
- How to transform one to other?
  - CONV_TAC, CONV_RULE

Rules, Conversions, Tactics

- Rule: `thm -> thm`
  - Examples?
- Conversion: `term -> thm`
  - Examples? `BETA_CONV`
- Tactic: goalstate refinement
  - Type? `goal -> goalstate`
- Thm_tactical: `thm_tactic -> thm_tactic`
- How to combine them?
  - THEN...
  - DEPTH_CONV
- How to transform one to other?
  - CONV_TAC, CONV_RULE

More advanced tactics

Tautologies

```
ITAUT `(A ==> B) ==> A ==> B`;;
TAUT `(p <=> (q <=> r)) <=> ((p <=> q) <=> r)`;;
```

Rewriting

- Matching
  - Slightly more general than first-order
- REWR_CONV does rewriting on top level

```
ADD_ASSOC;;
⊢ ∀mnp.m+n+p=(m+n)+p
REWR_CONV    ADD_ASSOC    `x+y+z`;;
⊢ x+y+z=(x+y)+z
```

- Conditional rewriting: REWRITE_RULE, REWRITE_CONV, REWRITE_TAC
- With basic rules: SIMP...

Arithmetic

Definition of Natural Numbers

From the axiom of infinity

Rewriting

- `REWRITE_TAC [ARITH]`
- What rules are being used?
- Is it complete?

Other domains

- Real, Complex, Integer, $\mathbb{R}^n$ (vectors)
- Bootstrapped decision procedures

What are Quotients

Basic notion in set theory (here HOL)

- Equivalence Relation (non-trivial)
  - Reflexivity, Symmetry, Transitivity
- Equivalence Classes
  - Give rise to a new type (quotient type)
  - Each element in the new type represents a class in the raw type
- Setoids in Coq

Standard Quotient Types

- Integers
  - Natural number pairs $nat \times nat$
  - Equivalence relation

    $$(n_1, n_2) \approx (m_1, m_2) \equiv n_1 + m_2 = m_1 + n_2$$

  - Zero and one as $(0, 0)$ and $(1, 0)$
  - Addition

    $$add\_pair\ (n_1, m_1)\ (n_2, m_2) \equiv (n_1 + n_2, m_1 + m_2)$$

- Finite Sets
  - Lists with the membership relation

    $$xs \approx ys \equiv (\forall x.\ memb\ x\ xs \longleftrightarrow memb\ x\ ys)$$

- Real numbers, Modular arithmetic, Multi-sets ...

Programming Language Calculi

- Lambda Calculus

$$
\begin{aligned}
t \;::=\; & x \\
\mid\; & t\ t \\
\mid\; & \lambda x.\ t
\end{aligned}
$$

Programming Language Calculi

- Lambda Calculus

$$
\begin{aligned}
t \ ::= \ & x \\
\mid \ & t \ t \\
\mid \ & \lambda x.\ t
\end{aligned}
$$

- With binders function definitions and proofs become complicated
  - defining substitution
  - substitution lemma

Programming Language Calculi

- Lambda Calculus

$$t ::= x \mid t\ t \mid \lambda x.\ t$$

- With binders function definitions and proofs become complicated
  - defining substitution
  - substitution lemma
- But $\lambda x.x \approx \lambda y.y$
- Quotienting makes such lemmas almost trivial
  - For simple binders done by Nominal/Isabelle

Isabelle/Nominal

- Framework for constructing α-equated terms

```
t ::= x
    | t t
    | λx. t        bind x in t
```

- Definitional extension of Isabelle/HOL
- Automatically derives a reasoning infrastructure
  - free variables (support and freshness)
  - renaming
  - strong induction principle

Isabelle/Nominal

Nominal has been used successfully in formalisations of:

- Equivalence checking algorithm for LF

  [*UrbanCheneyBerghofer08*]

- Typed Scheme

  [*TobinHochstadtFelleisen08*]

- Several calculi for concurrency

  [*BengtsonParow09*]

- Strong normalisation of cut-elimination in classical logic

  [*UrbanZhu08*]

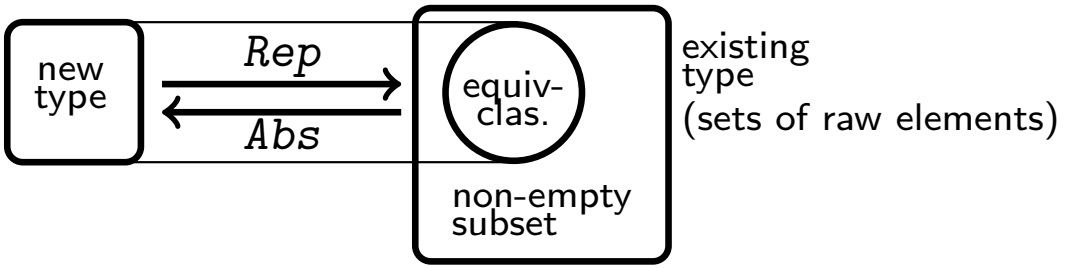
- Formalizations in the locally-nameless approach to binding

  [*SatoPollack10*]

What about multiple binders?

- We need automatic quotienting!

Defining the quotient type



- Subset of an existing type, can be defined as a new type
  - Type definition in HOL
- We choose representatives for each equivalence class
  - Choice based definitions of Abstraction and representation
  - (non-choice definitions possible without higher order)
- Can be a partial equivalence relation

[*Paulson01*]

Definitions in the quotient type

- Lifting constructors

  $0 \equiv Abs_{int}\ (0, 0)$ $\qquad$ $1 \equiv Abs_{int}\ (1, 0)$

- Lifting operations
  - Integer addition of type $int \Rightarrow int \Rightarrow int$:

    $add\ n\ m \equiv Abs_{int}\ (add\_pair\ (Rep_{int}\ n)\ (Rep_{int}\ m))$

- Polymorphic constants and higher order
  - Lifting $list\_fold$ to $fset\_fold$ [*Homeier05*]
- We extend the definitions for constants where the quotiented type occurs as a parameter:

  $flat\ [] \equiv []$ $\qquad$ $flat\ x::xs \equiv x\ @\ flat\ xs$

  - General algorithm for aggregate Abs and Rep

    $\bigcup\ S \equiv Abs_{fset}\ (flat\ ((map\_list\ Rep_{fset} \circ Rep_{fset})\ S))$

Lifting theorems

Similar to [*Homeier05*]

- Regularization
  - For types involving the raw type
    - Quantifiers and abstractions are replaced with bounded quantifiers and abstractions (respecting the quotient relation)
    - Equalities are replaced by equivalences
- Injection
  - For constants, application and abstractions that have their types changed a 'Abs-Rep' pair is introduced
- Cleaning
  - Rewriting the theorem with preservation rules for quantifiers, abstractions and constant definitions.

Prerequisites for lifting theorems

- Regularization
  - The only phase that can fail
  - If regularization fails, a stronger raw theorems is needed

$$1 \neq 0$$

$$\neg\ (1 \approx 0)$$

- Injection
  - Respectfulness rules for the constants

$$l1 \approx l2 \Longrightarrow (h :: l1) \approx (h :: l2)$$

- Cleaning
  - Preservation rules for the constants

```
map Abs (Rep x # map Rep xa) = x # xa
```

Compositional Quotients

- To lift

  $$\texttt{flat []} \equiv \texttt{[]} \qquad \texttt{flat x::xs} \equiv \texttt{x @ flat xs}$$

  to

  $$\bigcup \emptyset \equiv \emptyset \qquad \bigcup \texttt{\{x\}} \cup S \equiv x \cup \bigcup S$$

- We two quotient operations simultanously

  $$\alpha\ \texttt{list list} \Rightarrow \alpha\ \texttt{fset fset}$$

- We extend respectfulness and preservation
  - Many possible rules for one constructor
  - For list constructor

$$l1 \approx l2 \Longrightarrow (h :: l1) \approx (h :: l2)$$

$$l1\ (\approx \circ \texttt{list\_rel}\ \approx)\ l2 \Longrightarrow$$
$$(h :: l1)\ (\approx \circ \texttt{list\_rel}\ \approx)\ (h :: l2)$$

MESON

- Model-Elimination
  - Loveland 1968
- How it works
  - Given helper theorems (possibly polymorphic) assume them with appropriate types
  - Try to remove occurrences of the Hilbert operator
  - Eliminate trivial assumptions
  - Beta-reduce
  - Eliminate remaining abstractions (using λ-lifting)
  - Replace `if..then..else` expressions using Disjunctions
  - For quantification expressions over booleans, consider all cases
  - Transform to NNF and Skolemize
  - Make all applications first-order
  - Translate to FOL and execute model elimination

MESON export — monomorphisation

- Simple, but effective procedure
  - Find all polymorphic constants in the goal and the first assumption
  - For every occurrence of a constant in the goal and in the assumptions find a type instantiation
  - Apply the instantiation to the assumption and include its new instantiated constants in the goal constants
  - Repeat for all other assumptions
- May produce very big goals for set constants
- Considering all constants repeatedly can be very slow

MESON export — first order

- Given a term like:

  `MAP f [a] = [f a]`

  we have the symbol `f` sometimes applied to zero sometimes one argument
- Can be encoded in FO logic like:

  `MAP f [a] = [I f a]`

  If we assume that identity `I` is always applied to two arguments
- For every constant or free variable we find the minimum number of arguments it is applied to
- An application of a function `F` that needs two arguments to 4 arguments is now encoded as:

  `I (I (F(a1, a2), a3), a4)`

Looking at the code

- Unit type
- Quotient Package
- Pairs
- Natural numbers
- Inductive types
- Arithmetic
- Lists
- Reals
- Integers
- Sets

Summary

Today

- Natural numbers, datatypes
- Quotients, reals
- HOL Light library

Next time

- How hard is λ_P
- Second order logic
- Order of variables
- λ_2