Topics in Automated Deduction (CS 576)

Elsa L. Gunter

2112 Siebel Center

egunter@cs.uiuc.edu

http://www.cs.uiuc.edu/class/sp06/cs576/

Contact Information

- Office: 2233 Siebel Center
- Office hours: Tuesday 10:00 - 11:15 Thursday2:00 – 3:15
- Email: `egunter@cs.uiuc.edu`

Course Structure

- Text: Isabelle/HOL: A Proof Assistant for Higher-Order Logic

  by Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel

- Credit:
  - Homework (mostly submitted by email) 35%
  - Project and presentation 65%

- No Final Exam

Some Useful Links

- Website for class:

  `http://www.cs.uiuc.edu/class/sp06/cs576/`

- Website for Isabelle:

  `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/`

- Isabelle mailing list – to join, send mail to:

  `isabelle-users@cl.cam.ac.uk`

Text

- may be purchased: published by Springer Verlag as LNCS 2283

  http://www4.in.tum.de/~nipkow/LNCS2283/

- or may be downloaded locally:

  http://www.cs.uiuc.edu/class/sp06/cs576/doc/Isabelle-tutorial.pdf

- or directly for the main Isabelle website:

  http://www.cl.cam.ac.uk/Research/HVG/Isabelle/dist/Isabelle2004/doc/tutorial.pdf

Your Work

- Homework:
  - (Mostly) fairly short exercises carried out in Isabelle
  - Submitted by email
- Project:
  - Develop a model of a system in Isabelle
  - Prove some substantive properties of model
  - Discuss progress weekly in class
  - Give a 20 minute presentation of work at end of course

Course Objectives

- To learn to do formal reasoning
- To learn to model complex problems from computer science
- To learn to given fully rigorous proofs of properties

Crude Course Outline

- First Third: Introduction to Isabelle
  - Based on lecture notes by Tobias Nipow and by Larry Paulson
- Second Third: Jointly study a example of modeling and development of properties of model
- Last Third: Individual and small group development of projects
  - Projects may be of your proving, with my approval, or I will assign

Overview of Isabelle/HOL

System Architecture

ProofGeneral	(X)Emacs based interface
Isabelle/HOL	Isabelle instance for HOL
Isabelle	generic theorem prover
Standard ML	implementation language

HOL

- HOL = Higher-Order Logic
- HOL = Types + Lambda Calculus + Logic
- HOL has
  - datatypes
  - recursive functions
  - logical operators ($\wedge$, $\vee$, $\longrightarrow$, $\forall$, $\exists$, ...)
- HOL is very similar to a functional programming language
- Higher-order = functions are values, too!

Formulae (Approximation)

- Syntax (in decreasing priority):

$$
\begin{array}{rclcl}
form & ::= & (form) & | & term = term \\
 & | & \neg form & | & form \wedge form \\
 & | & form \vee form & | & form \longrightarrow form \\
 & | & \forall x.\ form & | & \exists x.\ form \\
 & & \text{and some others} & &
\end{array}
$$

- Scope of quantifiers: as for to right as possible

Examples

- $\neg A \wedge B \vee C \equiv ((\neg A) \wedge B) \vee C$
- $A \wedge B = C \equiv A \wedge (B = C)$
- $\forall x.\ P\ x \wedge Q\ x \equiv \forall x.\ (P\ x \wedge Q\ x)$
- $\forall x. \exists y.\ P\ x\ y \wedge Q\ x \equiv \forall x.(\exists y.\ (P\ x\ y \wedge Q\ x))$

Formulae

- Abbreviations:

  $\forall xy.\ P\ x\ y \equiv \forall x. \forall y.\ P\ x\ y \qquad (\forall, \exists, \lambda, \ldots)$

- Hiding and renaming:

  $\forall x\ y.\ (\forall x.\ P\ x\ y) \wedge Q\ x\ y \equiv \forall x_0\ y.(\forall x_1.P\ x_1\ y) \wedge Q\ x_0\ y$

- Parentheses:

  - $\wedge$, $\vee$, and $\longrightarrow$ associate to the right:

    $A \wedge B \wedge C \equiv A \wedge (B \wedge C)$

  - $A \longrightarrow B \longrightarrow C \equiv A \longrightarrow (B \longrightarrow C)$

    $\not\equiv (A \longrightarrow B) \longrightarrow C$ **!**

Warning!

Quantifiers have low priority (broad scope) and may need to be parenthesized:

$$! \quad \forall x.\ P\ x \wedge Q\ x \not\equiv (\forall x.\ Px) \wedge Q\ x \quad !$$

Types

Syntax:

τ	::=	(τ)	
	$\mid$	bool $\mid$ nat $\mid$ ...	base types
	$\mid$	'a $\mid$ 'b $\mid$ ...	type variables
	$\mid$	$\tau \Rightarrow \tau$	total functions (ascii : `=>`)
	$\mid$	$\tau \times \tau$	pairs (ascii : `*`)
	$\mid$	τ list	lists
	$\mid$	...	user-defined types

Parentheses: $T1 \Rightarrow T2 \Rightarrow T3 \equiv T1 \Rightarrow (T2 \Rightarrow T3)$

Terms: Basic syntax

Syntax:

$term$	$::=$	$(term)$	
	$\mid$	$c \mid x$	constant or variable (identifier)
	$\mid$	$term\ term$	function application
	$\mid$	$\lambda x.\ term$	function "abstraction"
	$\mid$	$\ldots$	lots of syntactic sugar

Examples: f (g x) y   h (λx. f (g x))

Parantheses: f a_1 a_2 $a_3 \equiv$ ((f a_1) a_2) a_3

Note: Formulae are terms

λ-calculus in a nutshell

Informal notation: $t[x]$

- **Function application:**

  $f\ a$ is the function f called with argument a.

- **Function abstraction:**

  $\lambda x.t[x]$ is the function with formal parameter x and body/result $t[x]$, i.e. $x \mapsto t[x]$.

λ-calculus in a nutshell

- **Computation:**

  Replace formal parameter by actual value

  ("β-reduction"): $(\lambda x.t[x])a \leadsto_\beta t[a]$

Example: $(\lambda x.\ x + 5)\ 3 \leadsto_\beta (3 + 5)$

Isabelle performs β-reduction automatically

Isabelle considers $(\lambda x.t[x])a$ and $t[a]$ equivalent

Terms and Types

Terms must be well-typed!

The argument of every function call must be of the right type

Notation: $t :: \tau$ maens t is a well-typed term of type τ

Type Inference

- Isabelle automatically computes (“infer”) the type of each variable in a term.
- In the presence of *overloaded* functions (functions with multiple, unrelated types) not always possible.
- User can help with type annotations inside the term.
- **Example:** f(x :: nat)

Currying

- **Curried:** f :: $\tau_1 \Rightarrow \tau_2 \Rightarrow \tau$
- **Tupled:** f :: $\tau_1 \times \tau_2 \Rightarrow \tau$

Advantage: partial appliaction f a_1 with $a_1 :: \tau$

Moral: Thou shalt curry your functions (most of the time :-)).

Terms: Syntactic Sugar

Some predefined syntactic sugar:

- Infix: $+$, $-$, $\#$, @, ...
- Mixfix: if_then_else_, case_of_, ...
- Binders: ∀x.P x means $(\forall)(\lambda x.\ P\ x)$

Prefix binds more strongly than infix:

$$!\quad \mathsf{f\ x + y} \equiv \mathsf{(f\ x) + y} \not\equiv \mathsf{f\ (x + y)} \quad !$$

Type bool

Formulae = terms of type bool

True::bool

False::bool

$\neg$ :: bool $\Rightarrow$ bool

$\wedge$, $\vee$, . . . :: bool $\Rightarrow$ bool

$\vdots$

if-and-only-if: $=$

Type nat

$0 :: nat$

$Suc :: nat \Rightarrow nat$

$+, *, \ldots :: nat \Rightarrow nat \Rightarrow nat$

$\vdots$

Overloading

! Numbers and arithmetic operations are overloaded:

0, 1, 2, . . . :: nat or real (or others)

$+ :: nat \Rightarrow nat \Rightarrow nat$ and

$+ :: real \Rightarrow real \Rightarrow real$ (and others)

You need type annotations: $1 :: nat$, $x + (y :: nat)$

. . . unless the context is unambiguous: Suc 0

Type list

- [ ]: empty list
- x # xs: list with first element x ("head") and rest xs ("tail")
- Syntactic sugar: $[x_1, \ldots, x_n] \equiv x_1 \# \ldots \# x_n \# [\,]$

Large library:

hd, tl, map,size, filter, set, nth, take, drop, distinct, ...

Don't reinvent, reuse!

$\leadsto$ `HOL/List.thy`