

CS576 Topics in Automated Deduction

Elsa L Gunter
2112 SC, UIUC
egunter@illinois.edu

<http://courses.engr.illinois.edu/cs576>

Slides based in part on slides by Tobias Nipow

January 21, 2015

Contact Information

- Office: 2112 SC
- Office Hours:
 - Fridays 11:00am - 12:15pm
 - Also by appointment
 - May add more if desirable
- Email: egunter@illinois.edu
- Newsgroup:
<https://piazza.com/illinois/spring2015/cs576/home>
- No TA this semester

Course Structure

- Recommended Texts:
 - Programming and Proving in Isabelle/HOL
by Tobias Nipkow
 - Isabelle/HOL: A Proof Assistant for Higher-Order Logic
by Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel
 - Concrete Semantics with Isabelle/HOL
Tobias Nipkow and Gerwin Klein,
<http://www.concrete-semantics.org>
- Credit:
 - Homework (submitted via svn) 33%
 - Project and presentation 67%
- No Final Exam

Some Useful Links

- Website for class:
<http://courses.engr.illinois.edu/cs576/sp2015/>
- 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

Your Work

- Homework:
 - (Mostly) fairly short exercises in Isabelle
 - Submitted via svn
- Project:
 - Develop a model of a system in Isabelle
 - Prove some substantive properties of model
 - Discuss progress weekly in class
 - Give 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 give fully rigorous proofs of properties

System Architecture

<i>Isabelle/jEdit</i>	jEdit based interface
<i>Isar</i>	Isabelle proof scripting language
<i>Isabelle/HOL</i>	Isabelle instance for HOL
<i>Isabelle</i>	generic theorem prover
<i>Standard ML</i>	implementation language

jEdit Input

Input of math symbols in jEdit

- via “standard” ascii name: `&`, `|`, `-->`, ...
- via ascii encoding (similar to $\text{\LaTeX}$):
`\<and>`, `\<or>`, ...
- via menu (“Symbols”)

Symbol Translations

symbol	$\forall$	$\exists$	λ	$\neg$	$\wedge$
ascii (1)	<code>\<forall></code>	<code>\<exists></code>	<code>\<lambda></code>	<code>\<not></code>	<code>\<and></code>
ascii (2)	ALL	EX	%	~	&

symbol	$\vee$	$\longrightarrow$	$\Rightarrow$
ascii (1)	<code>\<or></code>	<code>\<longrightarrow></code>	<code>\<Rightarrow></code>
ascii (2)		-->	=>

See Appendix A of tutorial for more complete list

Time for a demo of types and terms
(and a simple lemma)

Overview of Isabelle/HOL

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):

$form$	$::=$	$(form)$		$term = term$
		$\neg form$		$form \wedge form$
		$form \vee form$		$form \longrightarrow form$
		$\forall x. form$		$\exists x. form$

and some others

- Scope of quantifiers: as far to the right as possible

Examples

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

- Abbreviations:

$$\forall x y. P \ x \ y \equiv \forall x. \forall y. P \ x \ y \quad (\forall, \exists, \lambda, \dots)$$

- 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:

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

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

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

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:

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

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

Terms: Basic syntax

Syntax:

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

Examples: $f \ (g \ x) \ y \quad h \ (\lambda x. \ f \ (g \ x))$

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

Note: Formulae are terms

λ -calculus in a nutshell

Informal notation: $t[x]$

term t with 0 or more free occurrences of 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”): $(\lambda x. t[x])a \rightsquigarrow_{\beta} t[a]$

Example: $(\lambda x. x + 5) 3 \rightsquigarrow_{\beta} (3 + 5)$

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: $t :: \tau$ means t is well-typed term of type τ

Type Inference

- Isabelle automatically computes (“infers”) 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 application $f\ a_1$ with $a_1 :: \tau$

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

Terms: Syntactic Sugar

Some predefined syntactic sugar:

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

Prefix binds more strongly than infix:

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

Type bool

Formulae = terms of type bool

True::bool

False::bool

$\neg :: \text{bool} \Rightarrow \text{bool}$

$\wedge, \vee, \dots :: \text{bool} \Rightarrow \text{bool}$

$\vdots$

if-and-only-if: = but binds more tightly

Type nat

$0 :: \text{nat}$

$\text{Suc} :: \text{nat} \Rightarrow \text{nat}$

$+, \times, \dots :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$

$\vdots$

Overloading

! Numbers and arithmetic operations are overloaded:

$0, 1, 2, \dots :: \text{nat}$ or real (or others)

$+ :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$ and

$+ :: \text{real} \Rightarrow \text{real} \Rightarrow \text{real}$ (and others)

You need type annotations: $1 :: \text{nat}$, $x + (y :: \text{nat})$

... unless the context is unambiguous: $\text{Suc } 0$

Type list

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

List is supported by a large library:

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

Don't reinvent, reuse!

~> `HOL/List.thy`

A Recursive datatype

```
datatype 'a list = Nil    ("[]")  
                | Cons 'a "'a list" (infixr "#'" 65)
```

`[]`: empty list

`x # xs`: list with head `x::'a`, tail `xs::'a list`

A toy list: `False # (True # [])`

Syntactic sugar: `[False, True]`

Concrete Syntax

When writing terms and types in `.thy` files

Types and terms need to be enclosed in `"..."`

Except for single identifiers, e.g. `'a`

`" ..."` won't always be shown on slides

Structural Induction on Lists

$P\ xs$ holds for all lists xs if

- $P\ []$
- and for arbitrary y and ys , $P\ ys$ implies $P\ (y\ \# \ ys)$

$$\frac{\begin{array}{c} P\ ys \\ \vdots \\ P\ (y\ \# \ ys) \end{array}}{P\ xs}$$

A Recursive Function: List Append

Definition by *primitive recursion*:

```
primrec app :: "'a list  $\Rightarrow$  'a list  $\Rightarrow$  'a list
```

```
where
```

```
app [] ys = _____
```

```
app (x # xs) ys = _____app xs ..._____
```

One rule per constructor

Recursive calls only applied to constructor arguments

Guarantees termination (total function)

Demo: Append and Reverse