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 28, 2015

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:** $\mathrm{f(x::nat)}$

Currying

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

Advantage: partial application $\mathrm{f\ a_1}$ 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: $\forall x. P\ x$ 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

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

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

$\vdots$

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

Type nat

0::nat

Suc :: nat $\Rightarrow$ nat

$+$, $\times$, ... :: 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: $[x_1, \ldots, x_n] \equiv x_1 \# \ldots \# x_n \# [\,]$

List is supported be a large library:
hd, tl, map, size, filter, set, nth, take, drop, distinct, ...

Don't reinvent, reuse!
$\rightsquigarrow$ `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)`

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

A Recursive Function: List Append

Definition by *primitive recursion*:

```
primrec app :: "'a list ⇒ 'a list ⇒ '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

Proofs - Method 1

General schema:

lemma *name*: " ..."
apply (*method*)
⋮
done

If the lemma is suitable as a simplification rule:

lemma *name*[simp]: " ..."

Adds lemma *name* to future simplifications

Proof - Method 2

General schema:

lemma *lemma_name*: " ..."
proof (*method*)
fix *x y z*
assume *hyp1_name*: " ..."
from *hyp1_name*
show : " ..."
  proof *method*
  ⋮
  qed
qed

Will try to use only Method 2 (Isar) in lectures in class

Top-down Proofs

`sorry`

- "completes" any proof (by giving up, and accepting it)
- Suitable for top-down development of theories:
- Assume lemmas first, prove them later.

Only allowed for interactive proof!

Isabelle Syntax

- Distinct from HOL syntax
- Contains HOL syntax within it
- Also the same as HOL - need to not confuse them

Theory = Module

Syntax:

theory *MyTh*
imports $ImpTh_1$ ... $ImpTh_n$
begin

declarations, definitions, theorems, proofs, ...

end

- *MyTh*: name of theory being built. Must live in file *MyTh*`.thy`.
- $ImpTh_i$: name of *imported* theories. Importing is transitive.

Meta-logic: Basic Constructs

Implication: $\Longrightarrow$ (`==>`)
For separating premises and conclusion of theorems / rules
Equality: $\equiv$ (`==`)
For definitions
Universal Quantifier: $\bigwedge$ (`!!`)
Usually inserted and removed by Isabelle automatically

Do not use *inside* HOL formulae

Rule/Goal Notation

$$[\![A_1; \ldots ; A_n]\!] \Longrightarrow B$$

abbreviates

$$A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow B$$

and means the rule (or potential rule):

$$\frac{A_1; \ldots ; A_n}{B}$$

; $\approx$ "and"

Note: A theorem is a rule; a rule is a theorem.

The Proof/Goal State

1. $\bigwedge x_1 \ldots x_m.\ [\![A_1; \ldots ; A_n]\!] \Longrightarrow B$

$x_1 \ldots x_m$	Local constants (fixed variables)
$A_1 \ldots A_n$	Local assumptions
B	Actual (sub)goal

Proof Methods

- Simplification and a bit of logic

`auto` **Effect:** tries to solve as many subgoals as possible using simplification and basic logical reasoning

`simp` **Effect:** relatively intelligent rewriting with database of theorem, extra given theorems, and assumptions.

- More specialized tactics to come

Top-down Proofs

`sorry`

"completes" any proof (by giving up, and accepting it)
Suitable for top-down development of theories:
Assume lemmas first, prove them later.

Only allowed for interactive proof!