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 Nipkow

January 22, 2026

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: $\mathrm{if\_then\_else\_}$, $\mathrm{case\_of\_}$, ...
- Binders: $\forall \mathrm{x.P\ x}$ means $(\forall)(\lambda x.\ P\ x)$

Prefix binds more strongly than infix:

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

Type bool

Formulae = terms of type bool

True::bool

False::bool

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

$\wedge, \vee, \ldots :: \mathrm{bool} \Rightarrow \mathrm{bool}$

$\vdots$

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

Type nat

0::nat

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

$+, \times, \ldots :: \mathrm{nat} \Rightarrow \mathrm{nat} \Rightarrow \mathrm{nat}$

$\vdots$

Overloading

! Numbers and arithmetic operations are overloaded:

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

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

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

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

... 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: $[\mathrm{x_1}, \ldots, \mathrm{x_n}] \equiv \mathrm{x_1} \# \ldots \# \mathrm{x_n} \# [\,]$

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 ( ...)
  ⋮
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