Topics in Automated Deduction (CS 576)

Elsa L. Gunter

2112 Siebel Center

eguntercs.uiuc.edu

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

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: $\forall$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

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

$\vdots$

if-and-only-if: $=$

Type nat

0::nat

Suc :: nat $\Rightarrow$ nat

$+$, $*$, . . . :: 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`

Theory = Module

Syntax:

theory $MyTh$ **=** $ImpTh_1$**+** $\ldots$ **+**$ImpTh_n$**:**

(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.

Proof General

An Isabelle Interface

by David Aspinall

ProofGeneral

Customized version of (x)emacs:

- All of emacs (info: `Ctrl-h i`)
- Isabelle aware when editing `.thy` files
- (Optional) Can use mathematical symbols ("x-symbols")

Interaction:

- via mouse / buttons / pull-down menus
- or keybord (for key bindings, see `Ctrl-h m`)

ProofGeneral Input

Input of math symbols in ProofGeneral

- via menu ("X-Symbol")
- via ascii encoding (similar to LaTeX):

  `\<and>`, `\<or>`, ...

- via "standard" ascii name: `&`, `|`, `-->`, ...

Symbol Translations

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

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

(1) is converted to x-xymbol, (2) remains as ascii

See Appendix A of text for more complete list

Time for a demo of types and terms

A Recursive `datatype`

```
datatype 'a list = Nil | Cons 'a "'a list"
```

`Nil:` empty list

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

A toy list: Cons False (Cons True Nil)

Syntactic sugar: [False, True]

Contrete Syntax

When writing terms and types in `.thy` files (or an Isabelle shell):

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 Nil`
- and for arbitrary `y` and `ys`, `P ys` implies `P (Cons y ys)`

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

A Recursive Function: List Append

Declaration:

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

and definition by *primitive recursion*:

```
primrec
app Nil ys = ____
app (Cons 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

General schema:

```
lemma name: " ..."
apply ( ...)
⋮
done
```

If the lemma is suitable as a simplification rule:

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

Adds lemma *name* to future simplificaitons

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!