An Introduction to ProofPower

A Specification and Proof Tool
for Higher Order Logic

Course Objectives

- to describe the basic principles and concepts underlying ProofPower

- to enable the student to write simple specifications and undertake elementary proofs in HOL using ProofPower

- to enable the student to make effective use of the reference documentation

Course Outline

- Introduction
  - an overview of ProofPower
  - propositional and predicate calculus proofs
- Specification using ProofPower HOL
  - Primitive Syntax for TYPEs and TERMs
  - Derived Syntax for TYPEs and TERMs
  - Paragraphs (declarations) and Theories
- Proof in HOL
  - Basics of Proof
  - Rules, Conversions, Tactics...
  - Stripping, Rewriting
  - Induction

Course Prerequisites

Some familiarity with:

- first order predicate calculus

$\ulcorner(\forall x\bullet\ P\ x \Rightarrow R\ x) \Rightarrow ((\forall\ x\bullet\ P\ x) \Rightarrow (\forall x\bullet\ R\ x))\urcorner$;

- elementary set theory

$\ulcorner\forall a\ b\ c\bullet\ a \cap (b \cap c) = (a \cap b) \cap c\urcorner$;

- functional programming

SML

$\mathit{fun}\quad \mathit{fact}\ \mathit{0} = \mathit{1}$
$|\quad \mathit{fact}\ n = n * (\mathit{fact}\ (n - \mathit{1}))$;

Using Motif Window Manager

- After logging in type “openwin”.
- Use right mouse button away from windows or icons to get the **Root Menu**.
- Operate **menus** other than the Root Menu using the left mouse button.
- To **open** icon: single-click with left mouse button and use “Restore” menu item.
- To **close** window: single-click on menu button in top left corner and use “Minimize” menu item.
- To **move** window: single-click on menu button in top left corner and use “Move” menu item.
- To **resize** window: single-click on menu button in top left corner and use “Size” menu item.
- To **select** text: press left button at left of selection, drag pointer to right of selection and release button.
- To select **single line**: triple click with left button.
- To select **all text**: type Control-‘/’.
- To **copy and paste**: **select** source, press **copy** and, with pointer at destination, **paste**.

Using ProofPower

- Select **"HOL Course"** from the Root Menu to start up ProofPower for the course work.

- To **execute a command** enter it into the Script Window (upper text area), select it, and then use the Command Menu to "Execute Selection" (or type Control-X).

- Meta-language **prompt** is: ":>" in the Journal Window (lower text area).

- ML commands (or top level expressions) are **terminated by ";"** (use Control-; to add this if you forget).

- For short commands that you don't want to save in the script, use the **Command Line Tool**.

- Select Command Line Tool etc. from the **Tools Menu**

- In case of mismatching brackets or quotes you may get stuck with the **continuation prompt**: ": #'. In this case, use Command Menu to "Abandon" (or type Control-A).

- To enter **mathematical symbols**, use the **Palette Tool**. Get characters either by pressing the buttons (characters go in script window), or by drag-and-drop (character go to any text area).

- **Drag-and-drop** character by holding middle button over the character and dragging the pointer to target position; release button to drop character.

Exercises 0: Getting Started

1. Implement an ML function, *fact*, to compute factorials.

2. Test your solution; e.g. execute:

```
fact 0;
fact 1;
fact 6;
```

Hints:

- Iconified tools on right of the screen include a previewer for you to browse these slides and an xpp editor containing the source of the slides.

- Develop your solutions to the exercises in the xpp command session (tool on the left).

- Copy-and-paste material from the xpp editor where helpful.

Exercises 0: Solutions

The solution on slide 5 is fine, although it loops on negative numbers.

A more robust solution is:

SML

$$fun\ fact\ n = if\ n <= 0\ then\ 1\ else\ n * fact\ (n - 1);$$

Features of ProofPower

- Pedigree
- Power
- Assurance
- Openness
- Extensibility

Pedigree

- In tradition of Principia Mathematica.
- Based on Church's Simple Theory of Types.
- Milner style polymorphism
- Implementation builds on research at Universities of Edinburgh, Cambridge and Oxford.
- Follows "LCF paradigm".
- Metalanguage is Standard ML.

Power

ProofPower HOL is:

- Logically as expressive as Z.
- Notationally almost as concise as Z.
- Much less complex than Z.

ProofPower HOL has:

- 80% of the power of Z
for
20% of the complexity.
- Modern functional language, Standard ML, as "metalanguage", for carrying out proofs and programming extensions to the system.

Assurance

- Simple uncontroversial classical logical system.
- Mathematical and formal specifications of syntax and semantics of formal system.
- Good support for specification by conservative extension.
- Small (<10% system code) logical kernel, implemented as abstract datatype, enforces logical soundness of proofs.
- Formal specifications of logical kernel.

Openness

- support for standard well documented languages targetted:

  Standard ML, HOL, Z, SPARK

- most of the functions used to build system are available for re-use by the user

- comprehensive reference manual documenting all the functions supplied:

  $>$ 600 pages; $>$1000 ML names

- libraries of theories and "proof contexts" provided for re-use

Extensibility

- User has access to metalanguage (Standard ML) for:
  - developing proofs
  - extending system
  - domain specific proof automation
- extendible definitional forms
- customisable “proof contexts”
- designed to support multiple object languages
- parser generator available

Languages Supported

- NOW:
  - Standard ML (as metalanguage)
  - Higher Order Logic
  - Z
- SOON:
  - SPARK Annotation Language, via DRA's Compliance Notation
- EVENTUALLY (we hope):
  - ISO Standard Z
  - others

Functionality

- document preparation/printing:
  - using LaTeX "literate scripts" with extended fonts for document sources
  - indexes, cross reference and theory listings
- syntax check/type check (interactive or batch)
- formal reasoning (interactive or batch)
- theory management:
  - specifications and theorems held in theory hierarchy
  - programmable access to theory hierarchy

Levels of Use of ProofPower

- Education

  ProofPower is suitable for hands on interactive courses in mathematical logic, discrete mathematics and formal methods including Z. (however, course material needs to be developed)

- Specification

  ProofPower HOL can be used as a specification language without the need to understand the proof development facilities.

- Proof Development

  Most application proofs require knowledge of a modest subset of the proof facilities.

- Research/ Proof tool development

  ProofPower, like Cambridge HOL, is a good vehicle for research in a number of areas. Research, or other developments to the capabilities of the tool, can be undertaken by users, but requires deeper knowledge and understanding of the system.

Some Proofs are Easy with ProofPower

- propositional tautologies

  ProofPower proves these automatically, and uses propositional reasoning to simplify non-propositional goals automatically.

- first order predicate calculus

  Often these will also be automatically provable using resolution. Where resolution fails, there is a simple systematic approach to proving these results using ProofPower.

- elementary set theory

  A useful class of results from elementary set theory are automatically provable.

- other classes of results

  Whenever a new theory is introduced one or more proof contexts may be developed to solve automatically a range of results in that theory. "Decision procedures" for such classes of results can be made available via "prove_tac".

Simple Predicate Calculus Proofs

- use the subgoaling package

- set the goal

SML

```
set_goal([],⌜(∀x y• P x ⇒ R y)
        ⇔ (∀v w• ¬ P w ∨ R v)⌝);
```

- initiate proof by contradiction

SML

```
a contr_tac;
```

ProofPower output

```
Tactic produced 2 subgoals:
...
(* *** Goal "1" *** *)

(*  3 *)  ⌜∀ x y• P x ⇒ R y⌝
(*  2 *)  ⌜P w⌝
(*  1 *)  ⌜¬ R v⌝

(* ?⊢ *)  ⌜F⌝
```

- instantiate assumptions as required

SML

```
a (list_spec_asm_tac ⌜∀ x y• P x ⇒ R y⌝ [⌜w⌝,⌜v⌝]);
```


ProofPower output

$$
\begin{array}{l}
\textit{Tactic produced 0 subgoals}: \\
(*\ ***\ \textit{Goal}\ \texttt{"}\mathit{2}\texttt{"}\ ***\ *) \\
\\
(*\ \ \mathit{3}\ *)\ \ \ulcorner \forall\ v\ w \bullet \neg\ P\ w \lor R\ v \urcorner \\
(*\ \ \mathit{2}\ *)\ \ \ulcorner P\ x \urcorner \\
(*\ \ \mathit{1}\ *)\ \ \ulcorner \neg\ R\ y \urcorner \\
\\
(*\ ?\vdash\ *)\ \ \ulcorner F \urcorner
\end{array}
$$

SML

$$
a\ (list\_spec\_asm\_tac\ \ulcorner \forall\ v\ w \bullet \neg\ P\ w \lor R\ v \urcorner\ [\ulcorner y \urcorner, \ulcorner x \urcorner]);
$$

ProofPower output

$$
\begin{array}{l}
\textit{Tactic produced 0 subgoals}: \\
\textit{Current and main goal achieved}
\end{array}
$$

SML

$$
pop\_thm();
$$

ProofPower output

$$
\begin{array}{l}
\textit{Now 0 goals on the main goal stack} \\
val\ it = \vdash (\forall\ x\ y \bullet P\ x \Rightarrow R\ y) \\
\qquad\qquad\qquad \Leftrightarrow (\forall\ v\ w \bullet \neg\ P\ w \lor R\ v) : THM
\end{array}
$$

Exercises 1: Proof

Set the theory and the proof context:

SML

```
open_theory"hol";
set_pc "hol2";
```

Set the goal (from the examples supplied):

```
set_goal([],⌜conjecture⌝);
```

Then try the following methods of proof:

- Two tactic method using:

```
a contr_tac; (* once *)
a (list_spec_asm_tac ⌜asm⌝ [⌜t1⌝, ⌜t2⌝]);
  (* as many as necessary *)
```

- or

```
a (prove_tac[]); (* once *)
```

- or

```
a step_strip_tac; (* many times *)
```

in case of difficulty, revert to the two tactic method.

SML

```
(* Results from Principia Mathematica *2 *)
val PM2 =[
⌜(* *2.02 *) q ⇒ ( p ⇒ q)⌝,
⌜(* *2.03 *) (p ⇒ ¬ q) ⇒ (q ⇒ ¬ p)⌝,
⌜(* *2.15 *) (¬ p ⇒ q) ⇒ (¬ q ⇒ p)⌝,
⌜(* *2.16 *) (p ⇒ q) ⇒ (¬ q ⇒ ¬ p)⌝,
⌜(* *2.17 *) (¬ q ⇒ ¬ p) ⇒ (p ⇒ q)⌝,
⌜(* *2.04 *) (p ⇒ q ⇒ r) ⇒ (q ⇒ p ⇒ r)⌝,
⌜(* *2.05 *) (q ⇒ r) ⇒ (p ⇒ q) ⇒ (p ⇒ r)⌝,
⌜(* *2.06 *) (p ⇒ q) ⇒ (q ⇒ r) ⇒ (p ⇒ r)⌝,
⌜(* *2.08 *) p ⇒ p⌝,
⌜(* *2.21 *) ¬ p ⇒ (p ⇒ q)⌝];
```

SML

```
(* Results from Principia Mathematica *3 *)
val PM3 =[
(* *3.01 *) ⌜p ∧ q ⇔ ¬(¬ p ∨ ¬ q)⌝,
(* *3.2  *) ⌜p ⇒ q ⇒ p ∧ q⌝,
(* *3.26 *) ⌜p ∧ q ⇒ p⌝,
(* *3.27 *) ⌜p ∧ q ⇒ q⌝,
(* *3.3  *) ⌜(p ∧ q ⇒ r) ⇒ (p ⇒ q ⇒ r)⌝,
(* *3.31 *) ⌜(p ⇒ q ⇒ r) ⇒ (p ∧ q ⇒ r)⌝,
(* *3.35 *) ⌜(p ∧ (p ⇒ q)) ⇒ q⌝,
(* *3.43 *) ⌜(p ⇒ q) ∧ (p ⇒ r) ⇒ (p ⇒ q ∧ r)⌝,
(* *3.45 *) ⌜(p ⇒ q) ⇒ (p ∧ r ⇒ q ∧ r)⌝,
(* *3.47 *) ⌜(p ⇒ r) ∧ (q ⇒ s) ⇒ (p ∧ q ⇒ r ∧ s)⌝];
```


SML

```
(∗ Results from Principia Mathematica ∗4 ∗)
val PM4 =[
(∗ ∗4.1  ∗) ⌜p ⇒ q ⇔ ¬ q ⇒ ¬ p⌝,
(∗ ∗4.11 ∗) ⌜(p ⇔ q) ⇔ (¬ p ⇔ ¬ q)⌝,
(∗ ∗4.13 ∗) ⌜p ⇔ ¬¬ p⌝,
(∗ ∗4.2  ∗) ⌜p ⇔ p⌝,
(∗ ∗4.21 ∗) ⌜(p ⇔ q) ⇔ (q ⇔ p)⌝,
(∗ ∗4.22 ∗) ⌜(p ⇔ q) ∧ (q ⇔ r) ⇒ (p ⇔ r)⌝,
(∗ ∗4.24 ∗) ⌜p ⇔ p ∧ p⌝,
(∗ ∗4.25 ∗) ⌜p ⇔ p ∨ p⌝,
(∗ ∗4.3  ∗) ⌜p ∧ q ⇔ q ∧ p⌝,
(∗ ∗4.31 ∗) ⌜p ∨ q ⇔ q ∨ p⌝,
(∗ ∗4.33 ∗) ⌜(p ∧ q) ∧ r ⇔ p ∧ (q ∧ r)⌝,
(∗ ∗4.4  ∗) ⌜p ∧ (q ∨ r) ⇔ (p ∧ q) ∨ (p ∧ r)⌝,
(∗ ∗4.41 ∗) ⌜p ∨ (q ∧ r) ⇔ (p ∨ q) ∧ (p ∨ r)⌝,
(∗ ∗4.71 ∗) ⌜(p ⇒ q) ⇔ (p ⇔ (p ∧ q))⌝,
(∗ ∗4.73 ∗) ⌜q ⇒ (p ⇔ (p ∧ q))⌝];
```


SML

```
(* Results from Principia Mathematica *5 *)
val PM5 =[
(* *5.1  *) ⌜p ∧ q ⇒ (p ⇔ q)⌝,
(* *5.32 *) ⌜(p ⇒ (q ⇔ r)) ⇒ ((p ∧ q) ⇔ (p ∧ r))⌝,
(* *5.6  *) ⌜(p ∧ ¬ q ⇒ r) ⇒ (p ⇒ (q ∨ r))⌝];
```

SML

```
(* Definitions from Principia Mathematica *9 *)
val PM9 =[
(* *9.01 *) ⌜¬ (∀x• φx) ⇔ (∃x• ¬ φx)⌝,
(* *9.02 *) ⌜¬ (∃x• φx) ⇔ (∀x• ¬ φx)⌝,
(* *9.03 *) ⌜(∀x• φx ∨ p) ⇔ (∀x• φx) ∨ p⌝,
(* *9.04 *) ⌜p ∨ (∀x• φx) ⇔ (∀x• p ∨ φx)⌝,
(* *9.05 *) ⌜(∃x• φx ∨ p) ⇔ (∃x• φx) ∨ p⌝,
(* *9.06 *) ⌜p ∨ (∃x• φx) ⇔ p ∨ (∃x• φx)⌝];
val PM9b =[
(* *9.07 *) ⌜(∀x• φx) ∨ (∃y• ψy) ⇔ (∀x•∃y• φx ∨ ψy)⌝,
(* *9.08 *) ⌜(∃y• ψy) ∨ (∀x• φx) ⇔ (∀x•∃y• ψy ∨ φx)⌝];
```

SML

```
(* Results from Principia Mathematica *10 *)
val PM10 =[
(* *10.01  *) ⌜(∃x• φx) ⇔ ¬ (∀y• ¬ φy)⌝,
(* *10.1   *) ⌜(∀x• φx) ⇒ φy⌝,
(* *10.21  *) ⌜(∀x• p ⇒ φx) ⇔ p ⇒ (∀y• φy)⌝,
(* *10.22  *) ⌜(∀x• φx ∧ ψx) ⇔ (∀y• φy) ∧ (∀z• ψz)⌝,
(* *10.24  *) ⌜(∀x• φx ⇒ p) ⇔ (∃y• φy) ⇒ p⌝,
(* *10.27  *) ⌜(∀x• φx ⇒ ψx) ⇒ ((∀y• φy) ⇒ (∀z• ψz))⌝,
(* *10.271 *) ⌜(∀x• φx ⇔ ψx) ⇒ ((∀y• φy) ⇔ (∀z• ψz))⌝,
(* *10.28  *) ⌜(∀x• φx ⇒ ψx) ⇒ ((∃y• φy) ⇒ (∃z• ψz))⌝,
(* *10.281 *) ⌜(∀x• φx ⇔ ψx) ⇒ ((∃y• φy) ⇔ (∃z• ψz))⌝,
(* *10.35  *) ⌜(∃x• p ∧ φx) ⇔ p ∧ (∃y• φy)⌝,
(* *10.42  *) ⌜(∃x• φx) ∨ (∃y• ψy) ⇔ (∃z• φz ∨ ψz)⌝,
(* *10.5   *) ⌜(∃x• φx ∧ ψx) ⇒ (∃y• φy) ∧ (∃z• ψz)⌝,
(* *10.51  *) ⌜¬(∃x• φx ∧ ψx) ⇒ (∀y• φy ⇒ ¬ ψy)⌝];
```


SML

```
(* Results from Principia Mathematica *11 *)
val PM11 =[
(* *11.1  *) ⌜(∀x y• φ(x,y)) ⇒ φ(z,w)⌝,
(* *11.2  *) ⌜(∀x y• φ(x,y)) ⇔ ∀y x• φ(x,y)⌝,
(* *11.3  *) ⌜(p ⇒ (∀x y• φ(x,y)))
                    ⇔ (∀x y• p ⇒ φ(x,y))⌝,
(* *11.32 *) ⌜(∀x y• φ(x,y) ⇒ ψ(x,y))
                    ⇒ (∀x y• φ(x,y)) ⇒ (∀x y• ψ(x,y))⌝,
(* *11.35 *) ⌜(∀x y• φ(x,y) ⇒ p) ⇔ (∃x y• φ(x,y)) ⇒ p⌝,
(* *11.45 *) ⌜(∃x y• p ⇒ φ(x,y))
                    ⇔ (p ⇒ (∃x y• φ(x,y)))⌝,
(* *11.54 *) ⌜(∃x y• φx ∧ ψy) ⇔ (∃x• φx) ∧ (∃y• ψy)⌝,
(* *11.55 *) ⌜(∃x y• φx ∧ ψ(x,y))
                    ⇔ (∃x• φx ∧ (∃y• ψ(x,y)))⌝,
(* *11.6  *) ⌜(∃x• (∃y• φ(x,y) ∧ ψy) ∧ χx)
                    ⇔ (∃y• (∃x• φ(x,y) ∧ χx) ∧ ψy)⌝,
(* *11.62 *) ⌜(∀x y• φx ∧ ψ(x,y) ⇒ χ(x,y))
                    ⇔ (∀x• φx ⇒ (∀y• ψ(x,y) ⇒ χ(x,y)))⌝
];
```


SML

```
(* results from ZRM provable by stripping *)
val ZRM1 = [
⌜a ∪ a = a ∪ {}⌝,
⌜a ∪ {} = a ∩ a⌝,
⌜a ∩ a = a \ {}⌝,
⌜a \ {} = a⌝,
⌜a ∩ {} = a \ a⌝,
⌜a \ a = {} \ a⌝,
⌜{} \ a = {}⌝,
⌜a ∪ b = b ∪ a⌝,
⌜a ∩ b = b ∩ a⌝,
⌜a ∪ (b ∪ c) = (a ∪ b) ∪ c⌝,
⌜a ∩ (b ∩ c) = (a ∩ b) ∩ c⌝,
⌜a ∪ (b ∩ c) = (a ∪ b) ∩ (a ∪ c)⌝,
⌜a ∩ (b ∪ c) = (a ∩ b) ∪ (a ∩ c)⌝,
⌜(a ∩ b) ∪ (a \ b) = a⌝,
⌜(a \ b) ∩ b = {}⌝,
⌜a \ (b \ c) = (a \ b) ∪ (a ∩ c)⌝,
⌜(a \ b) \ c = (a \ (b ∪ c))⌝,
⌜a ∪ (b \ c) = (a ∪ b) \ (c \ a)⌝,
⌜a ∩ (b \ c) = (a ∩ b) \ c⌝,
⌜(a ∪ b) \ c = (a \ c) ∪ (b \ c)⌝];
```


SML

$val\ ZRM2 = [$
$\ulcorner a \setminus (b \cap c) = (a \setminus b) \cup (a \setminus c)\urcorner,$
$\ulcorner \neg\ x \in \{\}\urcorner,$
$\ulcorner a \subseteq a\urcorner,$
$\ulcorner \neg\ a \subset a\urcorner,$
$\ulcorner \{\} \subseteq a\urcorner,$
$\ulcorner \bigcup\ \{\} = \{\}\urcorner,$
$\ulcorner \bigcap\ \{\} = Universe\urcorner];$

SML

$(*\ results\ from\ ZRM\ *)$
$val\ ZRM3 = [$
$\ulcorner a \subseteq b \Leftrightarrow a \in \mathbb{P}\ b\urcorner,$
$\ulcorner a \subseteq b \land b \subseteq a \Leftrightarrow a = b\urcorner,$
$\ulcorner \neg\ (a \subset b \land b \subset a)\urcorner,$
$\ulcorner a \subseteq b \land b \subseteq c \Rightarrow a \subseteq c\urcorner,$
$\ulcorner a \subset b \land b \subset c \Rightarrow a \subset c\urcorner,$
$\ulcorner \{\} \subset a \Leftrightarrow \neg\ a = \{\}\urcorner,$
$\ulcorner \bigcup\ (a \cup b) = (\bigcup\ a) \cup (\bigcup\ b)\urcorner,$
$\ulcorner \bigcap\ (a \cup b) = (\bigcap\ a) \cap (\bigcap\ b)\urcorner,$
$\ulcorner\ a \subseteq b \Rightarrow \bigcup\ a \subseteq \bigcup\ b\ \urcorner,$
$\ulcorner\ a \subseteq b \Rightarrow \bigcap\ b \subseteq \bigcap\ a\ \urcorner];$

The HOL Type System

- abstract syntax/computation

SML

$$\begin{array}{l} mk\_vartype : string \quad\quad\quad\quad\quad\quad\quad\quad -> TYPE; \\ mk\_ctype : string * TYPE\ list \quad\quad\quad -> TYPE; \end{array}$$

- concrete syntax

BNF

$$\begin{array}{lcl} Type & = & Name \\ & | & Typars,\ Name \\ & | & Type,\ InfixName,\ Type \\ & | & `(`,\ Type,\ `)`; \\ Typars & = & Type \\ & | & `(`,\ Type,\ \{\ `,`,\ Type\ \},\ `)`; \end{array}$$

Type variables must begin with a prime.
Infix status and priority determined by fixity declarations.

- semantics
  - Types denote non-empty sets of values.
  - Type variables range over non-empty sets of values.
  - Type constructors denote functions from tuples of sets to sets.

Examples of Types

SML

```
⌜:'a⌝;
                        (* parsed type variable *)
val t = mk_vartype "'a";
                        (* computed type variable *)
val u = ⌜:BOOL⌝;
                        (* 0−ary type constructor *)
mk_ctype ("BOOL",[]);
                        (* computed 0−ary type construction *)
⌜:ℕ⌝;
                        (* 0−ary type constructor *)
⌜:'a LIST⌝;
                        (* polymorphic list type *)
⌜:(ℕ) LIST⌝;
                        (* lists of natural numbers *)
⌜:ℕ → ℕ⌝;
                        (* infix type constructor *)
mk_ctype ("→",[⌜:ℕ⌝,⌜:ℕ⌝]);
                        (* computed function space *)
⌜: ⌜SML: t⌝ → ⌜SML: u⌝⌝;
          (* another way of writing ⌜ML mk_ctype("→",[t,u])⌝ *)
⌜:ℕ × ℕ⌝;
                        (* pairs of natural numbers *)
⌜:ℕ + BOOL⌝;
                        (* disjoint union of ℕ and BOOL *)
⌜:(ℕ, ℕ) $×⌝;
                        (* suspending infix status *)
```


Computation with TYPEs (I) recognisers and destructors

- constructors

SML

$$mk\_vartype \;:string \qquad\qquad -> \; TYPE;$$
$$mk\_ctype \;:string * TYPE \; list \qquad -> \; TYPE;$$

- recognisers

SML

$$is\_vartype \;:TYPE \; -> \; bool;$$
$$is\_ctype \;:TYPE \; -> \; bool;$$

- destructors

SML

$$dest\_vartype \;:TYPE \; -> \; string;$$
$$dest\_ctype \;:TYPE \; -> \; string * TYPE \; list;$$

Computation with TYPEs (II)
sample functions

- type equality

SML

$op =: : TYPE * TYPE -> bool;$

- type variables in a type

SML

$type\_tyvars : TYPE -> string\ list;$

- type constructors in a type

SML

$type\_tycons : TYPE -> (string * int)\ list;$

- type instantiation

SML

$inst\_type : (TYPE * TYPE)\ list$
$\qquad\qquad -> TYPE -> TYPE;$

Computation with TYPEs (III) support for pattern matching

- DEST_SIMPLE_TYPE

```
datatype DEST_SIMPLE_TYPE =
            Vartype of string
|           Ctype of (string * TYPE list);
```

- generalised destructor

SML

```
dest_simple_type: TYPE -> DEST_SIMPLE_TYPE;
```

- generalised constructor

SML

```
mk_simple_type : DEST_SIMPLE_TYPE -> TYPE;
```

- pattern matching recursive functions

SML

```
fun type_tyvars2 t =
 (fn Vartype s        => [s]
 |   Ctype (s,tl)     => list_cup (map type_tyvars2 tl))
 (dest_simple_type t);
```


HOL Terms

- abstract syntax/computation

$$
\begin{array}{lllll}
\textit{datatype } DEST\_SIMPLE\_TERM = & & & & \\
 & & Var & of & string * TYPE \\
 & | & Const & of & string * TYPE \\
 & | & App & of & TERM * TERM \\
 & | & Simple\lambda & of & TERM * TERM;
\end{array}
$$

$$
\begin{array}{l}
dest\_simple\_term: TERM \; {-}{>} \; DEST\_SIMPLE\_TERM; \\
mk\_simple\_term: DEST\_SIMPLE\_TERM \; {-}{>} \; TERM;
\end{array}
$$

- concrete syntax

BNF

$$
\begin{array}{lll}
Term & = & \\
 & & `\lambda`,\ Name,\ [`{:}`,\ Type],\ `\bullet`,\ Term \\
 & | & Term,\ Term \\
 & | & Term,\ InfixName,\ Term \\
 & | & Term,\ `{:}`,\ Type \\
 & | & Name \\
 & | & `(`,\ Term,\ `)`;
\end{array}
$$

Names are treated as variables unless declared as constants.
Infix status and priority determined by fixity declarations.

Types of Terms

Terms must be well typed.
The type of a term is determined by type inference using the following rules:

- variables

$$\frac{}{\ulcorner v{:}\alpha \urcorner : \alpha}$$

- constants

$$\frac{}{\ulcorner c{:}\alpha \urcorner : \alpha}$$

- lambda abstractions

$$\frac{t : \alpha}{\ulcorner \lambda\ x{:}\beta \bullet t \urcorner : \beta \rightarrow \alpha}$$

- applications

$$\frac{f : \alpha \rightarrow \beta;\ x : \alpha}{\ulcorner f\ x \urcorner : \beta}$$

Types of Terms

The same rules may be rendered in ML as follows:

- variables

$$\frac{}{\mathit{type\_of}\ \ (\mathit{mk\_var}(\mathit{vname},\mathit{vtype}))\ =:\ \mathit{vtype};}$$

- constants

$$\frac{}{\mathit{type\_of}\ \ (\mathit{mk\_const}(\mathit{cname},\mathit{ctype}))\ =:\ \mathit{ctype};}$$

- lambda abstractions

$$\frac{\mathit{type\_of}\ \ \mathit{term}\ =:\ \mathit{ttype};}{\mathit{type\_of}\ \ulcorner\lambda\ x{:}'a \bullet {}_{\mathsf{ML}}\ulcorner \mathit{term}\urcorner\urcorner\ =:\ \ulcorner{:}'a \to \ulcorner_{SML:}\ \mathit{ttype}\urcorner\urcorner;}$$

- applications

$$\frac{\mathit{type\_of}\ \ \mathit{funterm}\ =:\ \ulcorner{:}'a \to 'b\urcorner; \qquad \mathit{type\_of}\ \ \mathit{arg}\ =:\ \ulcorner{:}'a\urcorner;}{\mathit{type\_of}\ \ulcorner{}_{\mathsf{ML}}\ulcorner \mathit{funterm}\urcorner\ {}_{\mathsf{ML}}\ulcorner \mathit{arg}\urcorner\urcorner\ =:\ \ulcorner{:}'b\urcorner;}$$

Types of Terms - Examples

SML

$$
\begin{array}{ll}
type\_of\ \ulcorner x{:}\mathbb{N}\urcorner & =: \ulcorner{:}\mathbb{N}\urcorner;\\
type\_of\ \ulcorner x{:}'a\urcorner & =: \ulcorner{:}'a\urcorner;\\
type\_of\ \ulcorner 0\urcorner & =: \ulcorner{:}\mathbb{N}\urcorner;\\
type\_of\ \ulcorner \lambda x{:}\mathbb{N} \bullet x + 1\urcorner & =: \ulcorner{:}\mathbb{N} \to \mathbb{N}\urcorner;\\
type\_of\ \ulcorner \lambda x \bullet x + 1\urcorner & =: \ulcorner{:}\mathbb{N} \to \mathbb{N}\urcorner;\\
type\_of\ \ulcorner (\lambda x \bullet x + 1)\ 3\urcorner & =: \ulcorner{:}\mathbb{N}\urcorner;\\
type\_of\ \ulcorner \$+\ 1\urcorner & =: \ulcorner{:}\mathbb{N} \to \mathbb{N}\urcorner;\\
type\_of\ \ulcorner \$+\urcorner & =: \ulcorner{:}\mathbb{N} \to \mathbb{N} \to \mathbb{N}\urcorner;
\end{array}
$$

$$
\begin{array}{ll}
type\_of\ \ulcorner T\urcorner & =: \ulcorner{:}BOOL\urcorner;\\
type\_of\ \ulcorner \neg\ T\urcorner & =: \ulcorner{:}BOOL\urcorner;\\
type\_of\ \ulcorner \$\neg\urcorner & =: \ulcorner{:}BOOL \to BOOL\urcorner;\\
type\_of\ \ulcorner \$\wedge\urcorner & =: \ulcorner{:}BOOL \to BOOL \to BOOL\urcorner;\\
type\_of\ \ulcorner \$\forall\urcorner & =: \ulcorner{:}('a \to BOOL) \to BOOL\urcorner;
\end{array}
$$

Semantics of Terms

- **Variables**

  range over the set denoted by their type.

- **Constants**

  denote particular values in the set denoted by their type.

- **Lambda Abstractions**

  denote total functions from the set denoted by the type of the variable to the set denoted by the type of the body.

  The value at point “p” is the value of the body when the variable is assigned value “p”.

- **Applications**

  denote the value of the function denoted by the first term at the point which is the value denoted by the second term.

Semantics of Terms - Examples

SML

$\beta\_conv\ \ulcorner(\lambda x \bullet x + 1)\ 3\urcorner;$

Hol Output

$val\ it = \vdash (\lambda\ x\bullet\ x + 1)\ 3 = 3 + 1 : THM$

SML

$rewrite\_conv[]\ \ulcorner(\lambda x \bullet x + 1)\ 3\urcorner;$

Hol Output

$val\ it = \vdash (\lambda\ x\bullet\ x + 1)\ 3 = 4 : THM$

SML

$\eta\_axiom;$

Hol Output

$val\ it = \vdash \forall\ f\bullet\ (\lambda\ x\bullet\ f\ x) = f : THM$

SML

$ext\_thm;$

Hol Output

$val\ it = \vdash \forall\ f\ g\bullet\ f = g \Leftrightarrow (\forall\ x\bullet\ f\ x = g\ x) : THM$

SML

$prove\_rule[]\ \ulcorner\exists\ x{:}\mathbb{N} \bullet\quad 43 = x\urcorner;$
$prove\_rule[]\ \ulcorner\exists\ b{:}BOOL \bullet\quad T = b\urcorner;$
$prove\_rule[]\ \ulcorner\forall\ x{:}\mathbb{N} \bullet\quad x \geq 0\urcorner;$
$prove\_rule[]\ \ulcorner\forall\ b{:}BOOL \bullet\quad b = T \lor b = F\urcorner;$

Derived Syntax - DEST_TERM

datatype **DEST_TERM =**	
DVar	*of string* $*$ *TYPE*
| **DConst**	*of string* $*$ *TYPE*
| **DApp**	*of TERM* $*$ *TERM*
| **D**λ	*of TERM* $*$ *TERM*
| **DEq**	*of TERM* $*$ *TERM*
| **D**$\Rightarrow$	*of TERM* $*$ *TERM*
| **DT**	
| **DF**	
| **D**$\neg$	*of TERM*
| **DPair**	*of TERM* $*$ *TERM*
| **D**$\wedge$	*of TERM* $*$ *TERM*
| **D**$\vee$	*of TERM* $*$ *TERM*
| **D**$\Leftrightarrow$	*of TERM* $*$ *TERM*
| **DLet**	*of* $((TERM * TERM) list * TERM)$
| **DEnumSet**	*of TERM list*
| **D**$\varnothing$	*of TYPE*
| **DSetComp**	*of TERM* $*$ *TERM*
| **DList**	*of TERM list*
| **DEmptyList**	*of TYPE*
| **D**$\forall$	*of TERM* $*$ *TERM*
| **D**$\exists$	*of TERM* $*$ *TERM*
| **D**$\exists_1$	*of TERM* $*$ *TERM*
| **D**ϵ	*of TERM* $*$ *TERM*
| **DIf**	*of* $(TERM * TERM * TERM)$
| **D**$\mathbb{N}$	*of int*
| **DChar**	*of string*
| **DString**	*of string*;

Derived Syntax

- prefix, infix and postfix operators
- binders
- pair matching lambda abstractions
- conditionals
- local definitions
- set displays and abstractions
- list displays
- literals (numeric, character, and string)

Binders

- Constants having type: ⌜:('a $\rightarrow$ 'b) $\rightarrow$ 'c⌝ (or any instance of this) may be declared as “binders”.

- Normally a “binder” is applied to a lambda expression, in which case the λ is omitted.

- binder status may be suspended by use of $.

SML

```
⌜∃ x• x = 4⌝ =$ ⌜$∃ λ x• x = 4⌝;
```

Nested Paired Abstractions

- nested lambda abstractions can be abbreviated as follows:

SML

$\ulcorner\lambda x{:}\mathbb{N}\bullet\lambda y{:}\mathbb{N}\bullet\ (x,y)\urcorner$ =\$ $\ulcorner\lambda\ x\ y{:}\mathbb{N}\bullet\ (x,y)\urcorner$;

This function takes two natural numbers and returns a pair. (“,” is the infix pairing operator.)

- functions taking pairs may be written:

SML

$rewrite\_conv$[] $\ulcorner(\lambda(x,y){:}\mathbb{N}\times\mathbb{N}\bullet\ x){=}Fst\urcorner$;

ProofPower output

$val\ it = \vdash (\lambda\ (x,\ y)\bullet\ x) = Fst \Leftrightarrow T : THM$

This function takes an argument which is an ordered pair, and returns the first element of the pair.

- these effects can be iterated or combined.

SML

$rewrite\_conv$ []
$\ulcorner(\lambda(x,y){:}\mathbb{N}\times\mathbb{N};\ ((v,w),z)\bullet\ x + y + v + w + z)$
$(1,2)\ ((3,4),5)\urcorner$;

ProofPower output

$val\ it =$
$\vdash (\lambda\ (x,\ y)\ ((v,\ w),\ z)\bullet\ x + y + v + w + z)$
$(1,\ 2)\ ((3,\ 4),\ 5) = 15 : THM$

Conditionals

- Conditionals may be written:

  if t1 **then** t2 **else** t3

SML

$rewrite\_conv[]\ \ulcorner if\ T\ then\ 0\ else\ 1\urcorner;$

ProofPower output

$val\ it = \vdash (if\ T\ then\ 0\ else\ 1) = 0 : THM$

SML

$rewrite\_conv[]\ \ulcorner if\ F\ then\ 0\ else\ 1\urcorner;$

ProofPower output

$val\ it = \vdash (if\ F\ then\ 0\ else\ 1) = 1 : THM$

SML

$rewrite\_conv[]\ \ulcorner if\ 3{>}6\ then\ x\ else\ y\urcorner;$

ProofPower output

$val\ it = \vdash (if\ 3 > 6\ then\ x\ else\ y) = y : THM$

Let Clauses (I)

- Local declarations may be made in the form:

  let defs **in** term

SML

$$rewrite\_conv[let\_def]\ulcorner let\ a = \texttt{"}Peter\texttt{"}\ in\ a,a\urcorner;$$

ProofPower output

$$\begin{aligned} val\ it = \vdash\ & (let\ a = \texttt{"}Peter\texttt{"}\ in\ (a,\ a)) \\ & = (\texttt{"}Peter\texttt{"},\ \texttt{"}Peter\texttt{"}) : THM \end{aligned}$$

- The left hand side of a definition may be a "varstruct":

SML

$$\begin{aligned} & rewrite\_conv[let\_def] \\ & \qquad \ulcorner let\ (x,y) = (1,T)\ in\ (y,x)\urcorner; \end{aligned}$$

ProofPower output

$$\begin{aligned} val\ it = \vdash\ & (let\ (x,\ y) = (1,\ T)\ in\ (y,\ x)) \\ & = (T,\ 1) : THM \end{aligned}$$

Let Clauses (II)

- The left hand side of a definition may be a function definition:

SML

$$rewrite\_conv[let\_def]\ulcorner let\ f\ x = x{*}x\ in\ f\ 3\urcorner;$$

ProofPower output

$$val\ it = \vdash (let\ f\ x = x * x\ in\ f\ 3) = 9 : THM$$

- Multiple definitions may be given in a single let clause.

SML

$$rewrite\_conv[let\_def]\ \ulcorner let\ a = 1\ and\ b = 2\ in\ (a,b)\urcorner;$$

ProofPower output

$$val\ it = \vdash (let\ a = 1\ and\ b = 2\ in\ (a,\ b)) = (1,\ 2) : THM$$

Set Displays

- Sets may be entered as terms by enumeration:

SML

$|rewrite\_conv[]\ulcorner 9 \in \{1*1;\ 2*2;\ 3*3;\ 4*4\}\urcorner;$

ProofPower Output

$|val\ it = \vdash 9 \in \{1 * 1;\ 2 * 2;\ 3 * 3;\ 4 * 4\}$
$|\qquad \Leftrightarrow T : THM$

SML

$|rewrite\_conv[]\ulcorner 10 \in \{1*1;\ 2*2;\ 3*3;\ 4*4\}\urcorner;$

ProofPower Output

$|val\ it = \vdash 10 \in \{1 * 1;\ 2 * 2;\ 3 * 3;\ 4 * 4\}$
$|\qquad \Leftrightarrow F : THM$

- Sets may also be entered as set abstractions:

SML

$|rewrite\_conv[]\ulcorner 9 \in \{x \mid x < 12\}\urcorner;$

ProofPower Output

$|val\ it = \vdash 9 \in \{x|x < 12\} \Leftrightarrow T : THM$

SML

$|rewrite\_conv[]\ulcorner z \in \{(x,\ y) \mid x < y\}\urcorner;$

ProofPower Output

$|val\ it = \vdash z \in \{(x,\ y)|x < y\}$
$|\qquad \Leftrightarrow Fst\ z < Snd\ z : THM$

List Displays

- A similar syntax is available for lists:

SML

$$rewrite\_conv[append\_def]$$
$$\ulcorner[1{*}1;\ 2{*}2;\ 3{*}3;\ 4{*}4]\ @\ [5{*}5]\urcorner;$$

ProofPower Output

$$val\ \ it = \vdash$$
$$[1 * 1;\ 2 * 2;\ 3 * 3;\ 4 * 4]\ @\ [5 * 5]$$
$$= [1;\ 4;\ 9;\ 16;\ 25] : THM$$

SML

$$\ulcorner Cons\ \ 1\ \ [2;3;4;5]\urcorner;$$

ProofPower Output

$$val\ \ it = \ulcorner[1;\ 2;\ 3;\ 4;\ 5]\urcorner : TERM$$

Literals (I)

- Numeric literals consist of a sequence of decimal digits (no sign):

SML

```
dest_simple_term ⌜123⌝;
```

ProofPower output

```
val it = Const ("123", ⓣℕ⌝)
        : DEST_SIMPLE_TERM
```

- Character literals consist of a single character in ‘ characters:

SML

```
dest_simple_term ⌜‘α‘⌝;
```

ProofPower output

```
val it = Const ("‘α", ⓣCHAR⌝) (* ‘ *)
        : DEST_SIMPLE_TERM
```

Literals (II)

- String literals consist of zero or more characters in “"” characters:

SML

$|$ *dest_simple_term* ⌜"*many characters* $\alpha\beta\gamma$"⌝;

ProofPower output

$|$ *val it* = *Const* ("\\"*many characters* $\alpha\beta\gamma$", (* " *)
$|$ ⌜:*CHAR LIST*⌝) : *DEST_SIMPLE_TERM*

- A string literal denotes a LIST of characters:

SML

$|$ *TOP_MAP_C string_conv* ⌜"*characters* $\alpha\beta\gamma$"⌝;

ProofPower output

$|$ *val it* = ⊢ "*characters* $\alpha\beta\gamma$"
$|$ = [‘c‘; ‘h‘; ‘a‘; ‘r‘; ‘a‘; ‘c‘; ‘t‘; ‘e‘; ‘r‘; ‘s‘;
$|$ ‘ ‘; ‘α‘; ‘β‘; ‘γ‘] : *THM*

Theories/Declarations/Definitions Specifications/Paragraphs

- Information about specifications is held in the theory database.

- The information is mainly put in the theories using various declarations and definitions, which are calls to ML functions.

- Some specifications may be effected using "paragraphs" in the object language (HOL).

Theories

A theory contains the following information:

- The name of the theory and the names of its parents and children.
- The names and arities of type constructors declared in the theory.
- The names and types of constants declared in the theory.
- Fixity and aliasing information.
- Definitions of constants.
- A collection of saved theorems.

Access to Theories

- To use a theory it must be "in context", this can be achieved be opening the theory or one of its descendents:

SML

$open\_theory : string -> unit;$

- To display the contents of a theory:

SML

$print\_theory : string -> unit;$

- To get things from the theory:

SML

$get\_aliases; get\_ancestors; get\_axiom; get\_axioms;$
$get\_binders; get\_children; get\_consts; get\_defn;$
$get\_defns; get\_descendants; get\_parents; get\_thm;$
$get\_thms; get\_spec;$

- To save things in the theory use declarations, definitions, specifications or paragraphs (see below).

Exercises 2: HOL Theory Explorations

- Find the names of all the theories:

SML

$get\_theory\_names()$;

- Print selected theories, e.g.:

SML

$open\_theory$ "$sets$";

$print\_theory$ "$sets$";

- Get the terms from the definitions in a theory, e.g.:

SML

$open\_theory$ "bin_rel";

$(map\ concl\ o\ map\ snd\ o\ get\_defns)$ "bin_rel";

- Now select interesting terms and take them apart using, e.g.:

SML

$dest\_simple\_term\ ⌜\forall\ r\ s\bullet\ r \oplus s = (Dom\ s \lhd r) \cup s⌝$;

Hol Output

$val\ it = App\ (⌜\$\forall⌝, ⌜\lambda\ r\bullet\ \forall\ s\bullet\ r \oplus s = (Dom\ s \lhd r) \cup s⌝)$

$: DEST\_SIMPLE\_TERM$

SML

$dest\_simple\_term\ ⌜\{1;2;3\}⌝$;

Hol Output

$val\ it = App\ (⌜Insert\ 1⌝, ⌜\{2;\ 3\}⌝) : DEST\_SIMPLE\_TERM$

SML

$get\_spec\ ⌜Insert⌝$;

Hol Output

$val\ it = \vdash \forall\ x\ y\ a$

$\bullet \neg x \in \{\} \land x \in Universe \land (x \in Insert\ y\ a \Leftrightarrow x = y \lor x \in a) : THM$

Declarations (I)

- theories and parents

SML

```
open_theory   : string -> unit;
new_theory    : string -> unit;
new_parent    : string -> unit;
```

- types

SML

```
new_type
 : string * int -> TYPE;
new_type_defn
 : string list * string * string list * THM -> THM;
declare_type_abbrev
 : string * string list * TYPE -> unit;
```

Declarations (II)

- constants

SML

```
new_const
  : string * TYPE -> TERM;
simple_new_defn
  : string list * string * TERM -> THM;
new_spec
  : string list * int * THM -> THM;
const_spec
  : string list * TERM list * TERM -> THM;
```

- types and constants

SML

```
unlabelled_product_spec;
          (* mainly for use with Z *)
labelled_product_spec;
          (* see paragraphs below *)
```

Declarations (III)

Any identifier can be declared:

- prefix, infix, postfix (with a priority)
- a binder (like "∀" and "∃")

SML

$$
\begin{array}{ll}
declare\_prefix & : int * string \mathrel{->} unit; \\
declare\_infix & : int * string \mathrel{->} unit; \\
declare\_postfix & : int * string \mathrel{->} unit; \\
declare\_binder & : string \mathrel{->} unit; \\
get\_fixity & : string \mathrel{->} Lex.FIXITY; \\
declare\_nonfix & : string \mathrel{->} unit;
\end{array}
$$

Paragraphs

Some declarations may be done without resort to the metalanguage:

- constant declarations (based on $const\_spec$)

SML

```
new_theory "tutorial";
declare_postfix (200, "!");
```

HOL Constant

$$\$! : \mathbb{N} \to \mathbb{N}$$

$$\begin{array}{lll} & & 0! = 1 \\ \land & \forall n{:}\mathbb{N}\bullet & (n+1)! = n! * (n+1) \end{array}$$

- labelled product declarations

HOL Labelled Product

$Date$

$$day\ month\ year{:}\mathbb{N}$$

Paragraphs - Example

HOL Constant

$$length : 'a\ LIST \to \mathbb{N}$$

$$length\ [] = 0$$
$$\land\ \forall\ h\ t \bullet$$
$$length\ (Cons\ h\ t) = length\ t + 1$$

SML

$$print\_theory\ "tutorial";$$
$$rewrite\_conv[get\_spec \ulcorner length \urcorner]$$
$$\ulcorner length\ [1;2;3;4;5] \urcorner;$$

Exercises 3: Specification

- Create a new theory as a child of "hol".

SML

```
open_theory "tutorial";
```

- Write a specification in HOL of a function to add the elements of a list of numbers.

  HINT: if your specification goes in as a "Constspec" then the system could not prove it consistent, and its probably either wrong or poorly structured. Try to make it clearly 'primitive recursive'.

- Use it to "evaluate" the term ⌜*list_sum*[1; 2; 3; 4; 5]⌝.

```
rewrite_conv[get_spec⌜list_sum⌝]
        ⌜list_sum [1;2;3;4;5]⌝;
```


Forward Proof in ProofPower

- **theorems** - values of type THM computed from **axioms** and **definitions** using **rules** and **conversions**.

- **axioms** - theorems introduced without proof.

- **definitions** - axioms introduced by "conservative" mechanisms.

- **rules** - functions which compute theorems.

- **conversions** - rules which prove equations from terms.

Theorems

- The **HOL logic** is a "sequent calculus".

- A **sequent** is a "(TERM list) * TERM" (=SEQ) where each term must have type ⌜:BOOL⌝.

- The list of TERMs are known as "assumptions" the single term is the conclusion of the sequent.

- A sequent is valid if whenever the assumptions are all true the conclusion is also true.

- A theorem is a sequent which has been derived from axioms and definitions using the rules of the logic. Theorems are tagged with an indicator of the context in which they were derived.

- The sequent part of a theorem may be accessed using:

  SML
  ```
  dest_thm : THM -> SEQ;
  asms     : THM -> TERM list;
  concl    : THM -> TERM;
  ```

- Theorems are displayed without "quine corners"; they cannot be parsed, they must be proven (or introduced as axioms).

- To see the primitive constants and axioms look in theories "min", "log" and "init".

Naming Conventions for Theorems and Rules

- *_axiom*
  ML names ending with *_axiom* are used for axioms and for functions (e.g. *new_axiom*) for introducing or accessing axioms.

- *_def _spec*
  ML name suffixes used for definitions.

- *_thm _clauses*
  ML name suffixes for theorems.

- *_rule _elim _intro*
  used for inference rules.

- *_conv*
  for conversions, rules having type TERM -> THM where the THM is an equation with the TERM as its left hand operand.

A Selection of Useful Rules (I)

- assume rule:

SML

$val\ thm1\ =\ asm\_rule\ ⌜\forall x\ y{:}\mathbb{N}\bullet\ x{*}y > 0⌝;$

ProofPower Output

$val\ thm1\ =\ \forall\ x\ y\bullet\ x * y > 0$
$\qquad\qquad\qquad\vdash\ \forall\ x\ y\bullet\ x * y > 0\ :\ THM$

- modus ponens

SML

$val\ thm\_a\ =\ asm\_rule⌜a{:}BOOL⌝;$
$val\ thm\_b\ =\ asm\_rule⌜a{\Rightarrow}b⌝;$

ProofPower Output

$val\ thm\_a\ =\ a \vdash a\ :\ THM$
$val\ thm\_b\ =\ a \Rightarrow b \vdash a \Rightarrow b\ :\ THM$

SML

$val\ thm\_c\ =\ {\Rightarrow}\_elim\ thm\_b\ thm\_a;$

ProofPower Output

$val\ thm\_c\ =\ a \Rightarrow b,\ a \vdash b\ :\ THM$

A Selection of Useful Rules (II)

- specialisation

SML

$val\ thm2 = \forall\_elim\ ⌜455⌝\ thm1;$

ProofPower Output

$val\ thm2 = \forall\ x\ y\bullet\ x * y > 0$
$\vdash \forall\ y\bullet\ 455 * y > 0 : THM$

- multiple specialisation

SML

$val\ thm3 = list\_\forall\_elim\ [⌜455⌝,⌜0⌝]\ thm1;$

ProofPower Output

$val\ thm3 = \forall\ x\ y\bullet\ x * y > 0$
$\vdash 455 * 0 > 0 : THM$

- removing outermost universals

SML

$val\ thm4 = all\_\forall\_elim\ thm1;$

ProofPower Output

$val\ thm4 = \forall\ x\ y\bullet\ x * y > 0 \vdash x * y > 0 : THM$

A Selection of Useful Rules (III)

- splitting conjunctions

SML

$val\ thm5\ =\ all\_\forall\_elim\ plus\_order\_thm;$

ProofPower output

$$\begin{array}{l} val\ thm5 = \vdash m + i = i + m \\ \qquad\qquad \land (i + m) + n = i + m + n \\ \qquad\qquad \land m + i + n = i + m + n : THM \end{array}$$

SML

$val\ thms1\ =\ strip\_\land\_rule\ thm5;$

ProofPower output

$$\begin{array}{l} val\ thms1 = [\vdash m + i = i + m, \\ \qquad\qquad \vdash (i + m) + n = i + m + n, \\ \qquad\qquad \vdash m + i + n = i + m + n] : THM\ list \end{array}$$

- adding universals (I)

SML

$val\ thm6\ =\ all\_\forall\_intro\ (nth\ 2\ thms1);$

ProofPower output

$val\ thm6 = \vdash \forall\ m\ i\ n \bullet m + i + n = i + m + n : THM$

- adding universals (II)

SML

$val\ thm7 = list\_\forall\_intro\ [\ulcorner i \urcorner, \ulcorner m \urcorner, \ulcorner n \urcorner]\ (nth\ 2\ thms1);$

ProofPower output

$val\ thm7 = \vdash \forall\ i\ m\ n \bullet m + i + n = i + m + n : THM$

Exercises 4: Forward Proof

1. Using $\Rightarrow\_elim$ and asm_rule prove:

   (a) $b \Rightarrow c, a \Rightarrow b, a \vdash c$

   (b) $a \Rightarrow b \Rightarrow c, a, b \vdash c$

2. Using $\forall\_elim$ with $\neg\_plus1\_thm$ prove:

   (a) $\vdash \neg 0 + 1 = 0$

   (b) $\vdash \neg x * x + 1 = 0$

3. Using $all\_\forall\_elim$ with $\leq\_trans\_thm$ prove:

   (a) $\vdash m \leq i \land i \leq n \Rightarrow m \leq n$

4. Using $list\_\forall\_elim$ prove:

   (a) (with $\neg\_less\_thm$)
   $\vdash \neg 0 < 1 \Leftrightarrow 1 \leq 0$

   (b) (with $\leq\_trans\_thm$)
   $\vdash \forall$ n• 3 $\leq$ x * x $\land$ x * x $\leq$ n $\Rightarrow$ 3 $\leq$ n

5. Using $all\_\forall\_elim$, $strip\_\land\_rule$, nth, $all\_\forall\_intro$:

   (a) (with $\leq\_clauses$)
   $\vdash \forall$ i m n• i + m $\leq$ i + n $\Leftrightarrow$ m $\leq$ n

   (b) (using $list\_\forall\_intro$)
   $\vdash \forall$ m i n• i + m $\leq$ i + n $\Leftrightarrow$ m $\leq$ n

Exercises 4: Solutions

SML

```
(* 1(a) *)
val ext1_thm1 = asm_rule ⌜a⇒b⌝;
val ext1_thm2 = asm_rule ⌜b⇒c⌝;
val ext1_thm3 = asm_rule ⌜a:BOOL⌝;
val ext1_thm4 = ⇒_elim ext1_thm1 ext1_thm3;
val ext1_thm5 = ⇒_elim ext1_thm2 ext1_thm4;
(* 1(b) *)
val ext2_thm1 =
 ⇒_elim (asm_rule ⌜a⇒b⇒c⌝)(asm_rule ⌜a:BOOL⌝);
```

SML

```
(* 2(a) *)
val ext3_thm1 = ∀_elim ⌜0⌝ ¬_plus1_thm;
(* 2(b) *)
val ext4_thm1 = ∀_elim ⌜x*x⌝ ¬_plus1_thm;
```

SML

```
(* 3(a) *)
val ext5_thm1 = all_∀_elim ≤_trans_thm;
```

SML

```
(* 4(a) *)
val ext6_thm1 = list_∀_elim [⌜0⌝,⌜1⌝] ¬_less_thm;
(* 4(b) *)
val ext7_thm1 = list_∀_elim [⌜3⌝,⌜x*x⌝] ≤_trans_thm;
```

SML

```
(* 5(a) *)
val ext8_thm1 = strip_∧_rule (all_∀_elim ≤_clauses);
val ext8_thm2 = all_∀_intro (nth 3 ext8_thm1);
(* 5(b) *)
val ext8_thm2 = list_∀_intro [⌜m⌝,⌜i⌝,⌜n⌝](nth 3 ext8_thm1);
```


Goal Oriented Proof

- a GOAL,

  is just a sequent, viz:

  - a list of assumptions (BOOLean TERMs)
  - a conclusion (also a BOOLean TERM)

  GOAL = TERM list * TERM = SEQ

- a PROOF,

  is a function which computes a theorem from a list of theorems.

  PROOF = THM list -> THM

- a TACTIC,

  is a function which:

  - takes a GOAL
  - returns
    - a list of sub-GOALs
    - a PROOF which will compute a theorem corresponding to ("achieving") the input goal from theorems corresponding to the sub-GOALs.

  TACTIC = GOAL -> (GOAL list * PROOF)

Using the Subgoal Package

- Getting started:

SML
```
set_goal : GOAL -> unit;
push_goal : GOAL -> unit;
push_consistency_goal : TERM -> unit;
```

- Moving along:

SML
```
apply_tactic : TACTIC -> unit;
a : TACTIC -> unit;
undo : int -> unit;
set_labelled_goal : string -> unit;
lemma_tac : TERM -> TACTIC;
```

- Finishing off:

SML
```
top_thm : unit -> THM;
pop_thm : unit -> THM;
save_pop_thm : string -> THM;
```

- also note:

SML
```
save_thm : string * THM -> THM;
list_save_thm
            : string list * THM -> THM;
save_consistency_thm
            : TERM -> THM -> THM;
```


Rewriting

$$\textbf{[pure\_][once\_][asm\_]rewrite\_}\begin{cases} conv \\ rule \\ tac \\ thm\_tac \end{cases}$$

$$: THM\ \ list- > \begin{cases} conv (= TERM- > THM) \\ THM- > THM \\ TACTIC \end{cases}$$

$$: THM- > TACTIC$$

rewrites the term, theorem or goal using:

- conversions in "proof context" (unless **pure**)
- assumptions (if **asm** but not **conv**) (after context sensitive pre-processing)
- theorems in $THM\ list$ (or THM) parameter (after context sensitive pre-processing)

Rewriting continues until no more rewrites are possible (unless **once**).

Exercises 5: Rewriting with the Subgoal Package

1. set a goal from the examples on set theory, e.g.:

SML

```
set_goal([],⌜a \ (b ∩ c) = (a \ b) ∪ (a \ c)⌝);
```

2. rewrite the goal using the current proof context:

SML

```
a (rewrite_tac[]);
```

3. step back using undo:

SML

```
undo 1;
```

4. now try rewriting without using the proof context:

```
a (pure_rewrite_tac[]);
```

   (this should fail)

Exercises 5 - Continued

5. try rewriting one layer at a time:

SML
$a\ (once\_rewrite\_tac[\,]);$

repeat until it fails.

6. now try rewriting with specific theorems:

SML
$set\_goal([\,],\ulcorner a \setminus (b \cap c) = (a \setminus b) \cup (a \setminus c)\urcorner);$
$a\ (pure\_rewrite\_tac[sets\_ext\_clauses]);$
$a\ (pure\_rewrite\_tac[set\_dif\_def]);$
$a\ (pure\_rewrite\_tac[\cap\_def,\ \cup\_def]);$
$a\ (pure\_rewrite\_tac[set\_dif\_def]);$

7. finish the proof by stripping:

SML
$a\ (REPEAT\ \ strip\_tac);$

8. extract the theorem

SML
$top\_thm();$

9. repeat the above then try repeating:

SML
$pop\_thm();$

Exercises 6: Combining Forward and Backward Proof

Prove the following results by rewriting using the goal package: for each example try the previous methods to see how they fail before following the hint

1. :

SML
```
set_goal([],⌜x + y = y + x⌝);
```

2. :

SML
```
set_goal([],⌜x + y + z = (x + y) + z⌝);
(* hint : try using plus_assoc_thm *)
```

3. :

SML
```
set_goal([],⌜z + y + x = y + z + x⌝);
(* hint : try using plus_assoc_thm1 *)
```

4. :

SML
```
set_goal([],⌜x + y + z = y + z + x⌝);
(* hint : try using ∀_elim with plus_assoc_thm1 *)
```

5. :

SML
```
set_goal([],⌜x + y + z + v = y + v + z + x⌝);
(* hint : try using ∀_elim with plus_order_thm *)
```


Exercises 6: Solutions

1. :

SML

```
set_goal([],⌜x + y = y + x⌝);
a (rewrite_tac[]);
```

2. :

SML

```
set_goal([],⌜x + y + z = (x + y) + z⌝);
a (rewrite_tac[plus_assoc_thm]);
```

3. :

SML

```
set_goal([],⌜z + y + x = y + z + x⌝);
a (rewrite_tac[plus_assoc_thm1]);
```

4. :

SML

```
set_goal([],⌜x + y + z = y + z + x⌝);
a (rewrite_tac[∀_elim ⌜y⌝ plus_assoc_thm1]);
```

5. :

SML

```
set_goal([],⌜x + y + z + v = y + v + z + x⌝);
a (rewrite_tac[∀_elim ⌜x⌝ plus_order_thm]);
```


Stripping

- "stripping" facilities incorporate automatic propositional reasoning and enable domain specific knowledge to be invoked automatically during proof.

- $strip\_tac$ processes the conclusion of the current goal

- When new assumptions are created, by $strip\_tac$ or otherwise, they are normally stripped before being entered into the assumption list.

- Stripping of assumptions is different from stripping of conclusions.

Stripping Conclusions (concl's)

1. apply conclusion stripping conversions from proof context

2. if no conversion applies then attempt one of the following:

   (a) :

   $$.. ?\vdash \forall x \bullet\ P\ x \ ===> \ .. ?\vdash P\ x'$$

   (b) :

   $$.. ?\vdash P1 \wedge P2 \ ===> \ .. ?\vdash P1\ and\ .. ?\vdash P2\ (two\ subgoals)$$

   (c) :

   $$.. ?\vdash P1 \Rightarrow P2 \ ===> \ strip\_asm\_tac(P1),\ .. ?\vdash P2$$

3. then check if:

   (a) conclusion of the goal is ⌜T⌝

   (b) conclusion is in the assumptions

   if so, prove the result

Stripping Assumptions (asm's)

1. Repeat the following transformations until no further changes occur: apply assumption stripping conversions from proof context
   (a) : apply assumption stripping conversions from proof context
   (b) :

   $$\exists x \bullet\ P\ x \vdash ?\ ..\ ===>\ P\ x' \vdash ?\ ..$$

   (c) :

   $$P1 \lor P2 \vdash ?\ ..\ ===>$$
   $$P1 \vdash ?\ ..\ and\ P2 \vdash ?\ ..\ (two\ subgoals)$$

   (d) :

   $$P1 \land P2 \vdash ?\ ..\ ===>$$
   $$P1,\ P2 \vdash ?\ ..\ (two\ assumptions)$$

2. then for each resulting assumption, check if:
   (a) assumption = F
   (b) assumption = concl
   (c) contradicts an existing assumption

   if so, prove the result.

3. also check if:
   (a) assumption = T
   (b) is same as an existing assumption

   if so, discard the assumption.

Exercises 7: Stripping

- Use the examples from Principia Mathematica & ZRM given earlier, e.g.:

SML

```
set_goal([],⌜p ∧ q ⇒ (p ⇔ q)⌝);
```

with

1. :

SML

```
a strip_tac;
```

2. and/or:

SML

```
a step_strip_tac;
```

- Observe the steps taken and try to identify the reasons for discharge of subgoals.

- Select the weakest "proof context":

SML

```
push_pc"initial";
```

then retry the examples (or previous exercises).

- When you have finished restore the original proof context by:

SML

```
pop_pc();
```


Induction

Induction principles can be expressed as theorems in Higher Order Logic, e.g.:

- $induction\_thm$

$$
\begin{aligned}
&\vdash \forall\ p\bullet \quad p\ 0 \quad \wedge \\
&\quad (\forall\ m\bullet\ p\ m \Rightarrow p\ (m + 1)) \\
&\Rightarrow \quad (\forall\ n\bullet\ p\ n) : THM
\end{aligned}
$$

- $cov\_induction\_thm$

$$
\begin{aligned}
&\vdash \forall\ p\bullet \quad (\forall\ n\bullet\ (\forall\ m\bullet\ m < n \Rightarrow p\ m) \Rightarrow p\ n) \\
&\Rightarrow \quad (\forall\ n\bullet\ p\ n) : THM
\end{aligned}
$$

- $list\_induction\_thm$

$$
\begin{aligned}
&\vdash \forall\ p\bullet \quad p\ [] \ \wedge \\
&\quad (\forall\ list\bullet\ p\ list \Rightarrow (\forall\ x\bullet\ p\ (Cons\ x\ list))) \\
&\Rightarrow \quad (\forall\ list\bullet\ p\ list) : THM
\end{aligned}
$$

Using $\forall\_elim$ and all_β_rule these can be specialised for use in forward proofs.

Induction Tactics

Special tactics are available to facilitate the use of induction principles:

- induction over natural numbers using *induction_tac*

$$\frac{\{\Gamma\}\ t}{\{\Gamma\}\ t[0/x];\ strip\{t,\ \Gamma\}\ t[x{+}1/x]} \qquad induction\_tac\ulcorner x\urcorner$$

- induction over natural numbers using *cov_induction_tac*

$$\frac{\{\Gamma\}\ t}{strip\{\ulcorner\forall m\bullet\ m < x \Rightarrow t[m/x]\urcorner,\ \Gamma\}\ t} \qquad cov\_induction\_tac\ulcorner x\urcorner$$

- induction over lists using *list_induction_tac*

$$\frac{\{\Gamma\}\ t}{\{\Gamma\}t[[]/x];\ strip\{t,\ \Gamma\}\ t[Cons\ h\ x/x]} \qquad list\_induction\_tac\ulcorner x\urcorner$$

Induction - Example (I)

Prove the associativity of append.

SML

```
set_goal([],⌜∀l1 l2 l3:'a LIST •
            (l1 @ l2) @ l3 = l1 @ (l2 @ l3)⌝);
(* remove universal quantifiers *)
a (REPEAT strip_tac);
```

ProofPower output

```
(* *** Goal "" *** *)

(* ?⊢ *)  ⌜(l1 @ l2) @ l3 = l1 @ l2 @ l3⌝
```

SML

```
(* induct on ⌜l1⌝ *)
a (list_induction_tac ⌜l1⌝);
```

ProofPower output

```
(* *** Goal "2" *** *)

(*  1 *)  ⌜(l1 @ l2) @ l3 = l1 @ l2 @ l3⌝

(* ?⊢ *)  ⌜∀ x• (Cons x l1 @ l2) @ l3
                 = Cons x l1 @ l2 @ l3⌝

(* *** Goal "1" *** *)

(* ?⊢ *)  ⌜([] @ l2) @ l3 = [] @ l2 @ l3⌝
```


Induction Example (II)

SML

```
a (rewrite_tac [append_def]);
```

ProofPower output

```
Tactic produced 0 subgoals:
Current goal achieved, next goal is:

(* *** Goal "2" *** *)

(*  1 *)  ⌜(l1 @ l2) @ l3 = l1 @ l2 @ l3⌝

(* ?⊢ *)  ⌜∀ x• (Cons x l1 @ l2) @ l3
                    = Cons x l1 @ l2 @ l3⌝
```

SML

```
a (asm_rewrite_tac [append_def]);
val append_assoc_thm = pop_thm();
```

ProofPower output

```
Tactic produced 0 subgoals:
Current and main goal achieved
val append_assoc_thm =
 ⊢ ∀ l1 l2 l3• (l1 @ l2) @ l3 = l1 @ l2 @ l3 : THM
```


Exercises 8: Induction

1. Appending the empty list has no effect:

SML

$$set\_goal([], \ulcorner \forall l1 \bullet l1 \mathbin{@} [] = l1 \urcorner);$$

2. “Reverse” distributes over “@” (sort of):

SML

$$set\_goal([], \ulcorner \forall l1\ l2 \bullet$$
$$\quad Rev\ (l1 \mathbin{@} l2) = (Rev\ l2) \mathbin{@} (Rev\ l1) \urcorner);$$

3. “Map” distributes over “@”:

SML

$$set\_goal([], \ulcorner \forall f\ l1\ l2 \bullet$$
$$\quad Map\ f\ (l1 \mathbin{@} l2) = (Map\ f\ l1) \mathbin{@} (Map\ f\ l2) \urcorner);$$

4. “Length” distributes over “@”:

SML

$$set\_goal([], \ulcorner \forall l1\ l2 \bullet Length\ (l1 \mathbin{@} l2)$$
$$\qquad = Length\ l1 + Length\ l2 \urcorner);$$

Exercises 8: Solutions

SML
```
set_goal([],⌜∀l1 • l1 @ [] = l1⌝);                (* no. 1 *)
a strip_tac;
a (list_induction_tac ⌜l1⌝
  THEN asm_rewrite_tac [append_def]);
val empty_append_thm = pop_thm();
```

SML
```
set_goal([],⌜∀l1 l2 • Rev (l1 @ l2) =
            (Rev l2) @ (Rev l1)⌝);(* no. 2 *)
a (REPEAT strip_tac);
a (list_induction_tac ⌜l1⌝ THEN asm_rewrite_tac
  [append_assoc_thm, empty_append_thm,
            append_def, rev_def]);
val rev_distrib_thm = pop_thm();
```

SML
```
set_goal([],⌜∀f l1 l2 • Map f (l1 @ l2) =
            (Map f l1) @ (Map f l2)⌝);        (* no. 3 *)
a (REPEAT strip_tac);
a (list_induction_tac ⌜l1⌝ THEN asm_rewrite_tac
  [map_def, empty_append_thm, append_def]);
val rev_distrib_thm = pop_thm();
```

SML
```
set_goal([],⌜∀l1 l2• Length (l1 @ l2) =
            Length l1 + Length l2⌝);          (* no. 4 *)
a (REPEAT strip_tac);
a (list_induction_tac ⌜l1⌝ THEN asm_rewrite_tac
  [append_def, length_def, plus_assoc_thm]);
val length_distrib_thm = pop_thm();
```


TACTICALs and other -ALs

- TACTICALs may be used to combine the available tactics.

- Expressions using TACTICALs may be used directly in proofs, e.g.:

```
a (REPEAT strip_tac);
```

- named tactics may be defined using TACTICALs:

SML
```
val repeat_strip_tac = REPEAT strip_tac;
```

- TACTICALs may be used to define parameterised tactics:

SML
```
fun list_induct_tac t = REPEAT strip_tac
                        THEN list_induction_tac t;
```

- tacticals usually have capitalised names ending in "_T", though the most common (e.g. REPEAT, THEN) have aliases omitting the "_T"

- other higher order functions are available:

  conversionals (_C suffix)
  THM_TACTICALs (_THEN suffix)
  THM_TACTICAL combinators (_TTCL suffix)

Commonly used TACTICALs

- REPEAT - takes a tactic and returns a tactic which repeats that tactic until it fails.

  If goal splits occur the repeating continues on all subgoals.

- THEN - an infix tactical which composes two tactics together. The second tactic is applied to all subgoals arising from the first tactic. If any applications of the operand tactics fail then the resulting tactic fails.

- ORELSE - an infix tactical which attempts to apply its first argument, and if this fails applies its second argument. If both arguments fail then the resulting tactic fails.

- TRY_T - a tactical taking one argument which will do nothing (but succeed!) if it argument tactic fails.

- THEN_TRY - variant on THEN which does not fail even if the second tactic fails.

  t1 THEN_TRY t2 = t1 THEN (TRY_T t2)

Exercises 9: TACTICALs

1. Write a tactic which does $strip\_tac$ three times.

   test it on:

SML

$$set\_goal([], \ulcorner (a \Rightarrow b \Rightarrow c) \Rightarrow a \Rightarrow b \Rightarrow c \urcorner);$$
$$set\_goal([], \ulcorner (a \Rightarrow b) \Rightarrow a \Rightarrow c \urcorner);$$

2. Write a tactic which does $strip\_tac$ up to 3 times.

   Try it on the same examples.

3. Write a tactic which takes two arguments:

   - a term which is a variable
   - a list of theorems

   and performs an inductive proof of a theorem concerning lists by:

   - stripping the goal
   - inducting on the variable
   - rewriting with the assumptions and the list of theorems

   Use it to shorten the earlier proofs about lists.

Exercises 9: Solutions

SML

```
(* no. 1 *)
val strip3_tac = strip_tac THEN strip_tac THEN strip_tac;
set_goal([],⌜(a ⇒ b ⇒ c) ⇒ a ⇒ b ⇒ c⌝);
a strip3_tac;
```

SML

```
(* no. 2 *)
val stripto3_tac = strip_tac THEN_TRY strip_tac
                             THEN_TRY strip_tac;
set_goal([],⌜(a ⇒ b) ⇒ a ⇒ c⌝);
a stripto3_tac;
```

SML

```
(* no. 3 *)
fun list_induct_tac var thl =
            REPEAT strip_tac
            THEN list_induction_tac var
            THEN_TRY asm_rewrite_tac thl;

set_goal([],⌜∀l1 l2 l3 •
  (l1 @ l2) @ l3 = l1 @ (l2 @ l3)⌝);
a (list_induct_tac ⌜l1:'a LIST⌝ [append_def]);
val append_assoc_thm = pop_thm ();

set_goal([], ⌜∀l1:'a LIST • l1 @ [] = l1⌝);
a (list_induct_tac ⌜l1:'a LIST⌝ [append_def]);
val empty_append_thm = pop_thm();
```


More Predicate Calculus Tactics (I)

strip_asm_tac

- **strip_asm_tac** strips a theorem into the assumptions in the same way that *strip_tac* adds new assumptions

Tactic

$$\frac{\{\ \Gamma\ \}\ t}{\{strip\ \ c,\ \Gamma\ \}\ t} \qquad \begin{array}{l} strip\_asm\_tac \\ (\vdash\ c) \end{array}$$

- a **case split** results if the conclusion of the theorem is a disjunction

- names ending in **_cases_thm** indicate theorems designed for use with *strip_asm_tac* for case splits:

$\mathbb{N}\_cases\_thm$	$\vdash \forall\ m \bullet\ m = 0 \lor (\exists\ i \bullet\ m = i + 1)$
$less\_cases\_thm$	$\vdash \forall\ m\ n \bullet\ m < n \lor m = n \lor n < m$

- use [*list_*]$\forall$*_elim* to specialise the *_cases_thm*

$strip\_asm\_tac$: example

SML

```
set_goal([], ⌜(if x = 0 then 1 else x) > 0⌝);
```

SML

```
∀_elim⌜x⌝ℕ_cases_thm;
```

ProofPower Output

```
val it = ⊢ x = 0 ∨ (∃ i• x = i + 1) : THM
```

SML

```
a(strip_asm_tac(∀_elim⌜x⌝ℕ_cases_thm));
```

ProofPower Output

```
(* *** Goal "2" *** *)

(*  1 *)  ⌜x = i + 1⌝

(* ?⊢ *)  ⌜(if x = 0 then 1 else x) > 0⌝

(* *** Goal "1" *** *)

(*  1 *)  ⌜x = 0⌝

(* ?⊢ *) ⌜(if x = 0 then 1 else x) > 0⌝
```


More Predicate Calculus Tactics (II)

cases_tac

- **cases_tac**$\ulcorner c \urcorner$ lets you reason by cases according as a chosen condition c is true or false:

Tactic

$$\frac{\{\ \Gamma\ \}\ t}{\{strip\ \ c,\ \Gamma\ \}\ t;\quad \{strip\ \neg c,\ \Gamma\ \}\ t} \qquad \begin{array}{l} cases\_tac \\ \ulcorner c{:}BOOL \urcorner \end{array}$$

- $cases\_tac \ulcorner c : BOOL \urcorner$ is effectively the same as:

$strip\_asm\_tac(\forall\_elim \ulcorner c{:}BOOL \urcorner (prove\_rule[] \ulcorner \forall b \bullet b \lor \neg b \urcorner));$

but it's less to type and quicker.

cases_tac: example

SML

```
set_goal([], ⌜(if x < y + 1 then x else y) < y + 1⌝);
```

SML

```
a(cases_tac ⌜x < y + 1⌝);
```

ProofPower Output

```
(* *** Goal "2" *** *)

(*  1 *)  ⌜¬ x < y + 1⌝

(* ?⊢ *)  ⌜(if x < y + 1 then x else y) < y + 1⌝

(* *** Goal "1" *** *)

(*  1 *)  ⌜x < y + 1⌝

(* ?⊢ *)  ⌜(if x < y + 1 then x else y) < y + 1⌝
```


More Predicate Calculus Tactics (III)

swap_asm_concl_tac

- **swap_asm_concl_tac** lets you interchange (the negations) of an assumption and a conclusion

Tactic

$$\frac{\{\ \Gamma,\ t1\ \}\ t2}{\{strip\ \neg t2,\ \Gamma\ \}\ \neg t1} \qquad \begin{array}{l} swap\_asm\_concl\_tac \\ \ulcorner t1 \urcorner \end{array}$$

- Often used to rewrite one assumption with another

- Also useful when the conclusion is a negated equation

$swap\_asm\_concl\_tac$: example

SML

$set\_goal([], \ulcorner(\forall x\ y\bullet x \le y \Rightarrow P(x,y)) \land x = y \Rightarrow P(x,y)\urcorner);$
$a(strip\_tac);$

ProofPower Output

$(*\ \ 2\ *)\ \ \ulcorner\forall\ x\ y\bullet\ x \le y \Rightarrow P\ (x,\ y)\urcorner$
$(*\ \ 1\ *)\ \ \ulcorner x = y\urcorner$

$(*\ ?\vdash\ *)\ \ \ulcorner P\ (x,\ y)\urcorner$

ProofPower Output

SML

$a(list\_spec\_nth\_asm\_tac\ 2\ [\ulcorner x\urcorner, \ulcorner y\urcorner]);$

ProofPower Output

$(*\ \ 3\ *)\ \ \ulcorner\forall\ x\ y\bullet\ x \le y \Rightarrow P\ (x,\ y)\urcorner$
$(*\ \ 2\ *)\ \ \ulcorner x = y\urcorner$
$(*\ \ 1\ *)\ \ \ulcorner\neg\ x \le y\urcorner$

$(*\ ?\vdash\ *)\ \ \ulcorner P\ (x,\ y)\urcorner$

SML

$a(swap\_asm\_concl\_tac\ \ulcorner\neg\ x \le y\urcorner);$

ProofPower Output

$(*\ \ 3\ *)\ \ \ulcorner\forall\ x\ y\bullet\ x \le y \Rightarrow P\ (x,\ y)\urcorner$
$(*\ \ 2\ *)\ \ \ulcorner x = y\urcorner$
$(*\ \ 1\ *)\ \ \ulcorner\neg\ P\ (x,\ y)\urcorner$

$(*\ ?\vdash\ *)\ \ \ulcorner x \le y\urcorner$

More Predicate Calculus Tactics (IV)

lemma_tac

- **lemma_tac** lets you state and prove an "in-line" lemma

Tactic

$$\frac{\{\ \Gamma\ \}\ conc}{\{\ \Gamma\ \}\ lemma; \quad \{strip\ lemma,\ \Gamma\ \}\ conc} \qquad \begin{array}{l} lemma\_tac \\ \ulcorner lemma \urcorner \end{array}$$

- Gives a more natural feel to many proofs
- If just one tactic will prove the lemma then **THEN1** is a convenient way of applying it
- *tac1 THEN1 tac2* first applies *tac1* and then applies *tac2* to the first resulting subgoal

lemma_tac: example

SML

```
set_goal([], ⌜(∀x y•x ≤ y ⇒ P(x,y)) ∧ x = y ⇒ P(x,y)⌝);
a(strip_tac);
```

ProofPower Output

```
(*  2 *)  ⌜∀ x y• x ≤ y ⇒ P (x, y)⌝
(*  1 *)  ⌜x = y⌝

(* ?⊢ *)  ⌜P (x, y)⌝
```

SML

```
a(lemma_tac⌜x ≤ y⌝);
```

ProofPower Output

```
(* *** Goal "2" *** *)
(*  3 *)  ⌜∀ x y• x ≤ y ⇒ P (x, y)⌝
(*  2 *)  ⌜x = y⌝
(*  1 *)  ⌜x ≤ y⌝

(* ?⊢ *)  ⌜P (x, y)⌝

(* *** Goal "1" *** *)
(*  2 *)  ⌜∀ x y• x ≤ y ⇒ P (x, y)⌝
(*  1 *)  ⌜x = y⌝

(* ?⊢ *)  ⌜x ≤ y⌝
```


Processing of “New” Assumptions

- Tactics which add new assumptions normally do so with $strip\_asm\_tac$.

  E.g., $strip\_tac$, $cases\_tac$, $lemma\_tac$ work like this.

  Sometimes, this causes more case splitting than you might expect.

- if xxx_tac adds new assumptions, then often XXX_T exists to allow the new assumption to be used some other way.

- commonly, XXX_T has an argument of type $THM->TACTIC$ telling what to do with the new assumption.

  E.g., $cases\_tac$ is the same as $CASES\_T$ $strip\_asm\_tac$.

- Other useful $THM->TACTICS$ include:

$asm\_tac(\vdash t)$	put t into the assumptions as is (good for debugging)
$ante\_tac(\vdash$ t)	conclusion, c, becomes $t \Rightarrow c$
$rewrite\_thm\_tac(\vdash$ t)	rewrite with $\vdash t$

Take care with $rewrite\_thm\_tac$: it discards the new assumption after rewriting with it. It’s safe in examples like:

SML

```
set_goal([], ⌜(if x < y + 1 then x else y) < y + 1⌝);
a(CASES_T ⌜x < y + 1⌝ rewrite_thm_tac);
```

ProofPower Output

```
Tactic produced 0 subgoals:
Current and main goal achieved
```


Exercises 10: $strip\_asm\_tac$ etc.

1. Use $strip\_asm\_tac$ (with $\forall\_elim$ and $\mathbb{N}\_cases\_thm$) or $cases\_tac$ to prove

   (a) $\forall x \bullet (if\ x = 0\ then\ 1\ else\ x) > 0$

   (b) $\forall x\ y \bullet (if\ x < y + 1\ then\ x\ else\ y) < y + 1$

   (c) $\forall a\ b \bullet a \leq (if\ a \leq b\ then\ b\ else\ a)$

   (d) $\forall a \bullet a = 0 \lor 1 \leq a$

2. Using *(i)* $swap\_asm\_concl\_tac$ and *(ii)* $lemma\_tac$ give two different proofs of each of:

   (a) $(\forall x\ y \bullet x \leq y \Rightarrow\ P(x,\ y)) \Rightarrow (\forall x\ y \bullet x = y \Rightarrow\ P(x,\ y))$

   (b) $(\forall x\ y \bullet f\ x \leq f\ y \Rightarrow\ x \leq y) \Rightarrow (\forall x\ y \bullet f\ x = f\ y \Rightarrow x \leq y)$

Exercises 10/1 : Solutions

SML

```
                                                        (* (a) *)
set_goal([], ⌜∀x•(if x = 0 then 1 else x) > 0⌝);
a(REPEAT strip_tac);
a(strip_asm_tac(∀_elim⌜x⌝ℕ_cases_thm) THEN asm_rewrite_tac[]);
pop_thm();
```

SML

```
                                                        (* (b) *)
set_goal([], ⌜∀x y•(if x < y + 1 then x else y) < y + 1⌝);
a(REPEAT strip_tac);
a(CASES_T ⌜x < y + 1⌝ rewrite_thm_tac);
pop_thm();
```

SML

```
                                                        (* (c) *)
set_goal([], ⌜∀a b•a ≤ (if a ≤ b then b else a)⌝);
a(REPEAT strip_tac);
a(CASES_T ⌜a ≤ b⌝ rewrite_thm_tac);
pop_thm();
```

SML

```
                                                        (* (d) *)
set_goal([], ⌜∀a•a = 0 ∨ 1 ≤ a⌝);
a(strip_tac);
a(strip_asm_tac(∀_elim⌜a⌝ℕ_cases_thm) THEN asm_rewrite_tac[]);
pop_thm();
```


Exercises 10/2 : Solutions

With *swap_asm_concl_tac*:

SML

```
set_goal([],                                          (* (i)(a) *)
⌜(∀x y•x ≤ y ⇒  P(x, y)) ⇒ (∀x y•x = y ⇒  P(x, y))⌝);
a(REPEAT strip_tac);
a(list_spec_nth_asm_tac 2[⌜x⌝, ⌜y⌝]);
a(swap_asm_concl_tac ⌜¬ x ≤ y⌝ THEN asm_rewrite_tac[]);
pop_thm();
```

SML

```
set_goal([],                                          (* (i)(b) *)
⌜(∀x y•f x ≤ f y ⇒  x ≤ y) ⇒ (∀x y•f x = f y ⇒ x ≤ y)⌝);
a(REPEAT strip_tac);
a(list_spec_nth_asm_tac 2[⌜x⌝, ⌜y⌝]);
a(swap_asm_concl_tac ⌜¬ f x ≤ f y⌝ THEN asm_rewrite_tac[]);
pop_thm();
```

With *lemma_tac*:

SML

```
set_goal([],                                          (* (ii)(a) *)
⌜(∀x y•x ≤ y ⇒  P(x, y)) ⇒ (∀x y•x = y ⇒  P(x, y))⌝);
a(REPEAT strip_tac);
a(lemma_tac⌜x ≤ y⌝ THEN1 asm_rewrite_tac[]);
a(list_spec_nth_asm_tac 3 [⌜x⌝, ⌜y⌝]);
pop_thm();
```

SML

```
set_goal([],                                          (* (ii)(b) *)
⌜(∀x y•f x ≤ f y ⇒  x ≤ y) ⇒ (∀x y•f x = f y ⇒ x ≤ y)⌝);
a(REPEAT strip_tac);
a(lemma_tac⌜f x ≤ f y⌝ THEN1 asm_rewrite_tac[]);
a(list_spec_nth_asm_tac 3 [⌜x⌝, ⌜y⌝]);
pop_thm();
```


Forward Chaining (I)

- **Forward chaining** refers to a group of tactics for reasoning forward from the assumptions.

- Based on a rule, **fc_rule**, which uses a list of implications to generate a list of new theorems from a list of "seed" theorems. Arguments are two lists:

Implications	maybe universally quantified: $[\Gamma_1 \vdash \forall x1\ x2\ ...\bullet A_1 \Rightarrow B_1,\ ...]$
Seeds	any form: $[\Gamma_1 \vdash c_1,\ ...]$

- For each implication, $\vdash \forall x1\ x2\ ...\bullet A \Rightarrow B$ and for each seed $\vdash c$, *fc_rule* determines whether A can be specialised to give c and if so it includes the corresponding specialisation of B in its result. For example:

SML

```
(fc_rule : THM list -> THM list -> THM list)
        [asm_rule⌜∀x•x > 10 ⇒ P x⌝,
         asm_rule⌜∀y•y < 10 ⇒ Q y⌝]
        [prove_rule[]⌜101 > 10⌝,
         prove_rule[]⌜4 < 10⌝];
```

ProofPower Output

```
val it = [∀ y• y < 10 ⇒ Q y ⊢ Q 4,
        ∀ x• x > 10 ⇒ P x ⊢ P 101] : THM list
```


Forward Chaining (II)

- In practice, don't call fc_rule directly. Instead use one of the forward chaining tactics:

$$\textbf{[all\_][asm\_]fc\_tac}$$

$$\textbf{[all\_][asm\_]forward\_chain\_tac}$$

- All have type

$$\textbf{THM list} -> \textbf{TACTIC}$$

- $asm\_$ variants take implications to be the argument together with the assumptions. Other variants just use list given as argument. In all cases the seeds are the assumptions.

- Variants without $all\_$ take one pass over the seeds for each implication. Variants with $all\_$ add any new implications to the list of implications and loop until no new results can be generated.

- New theorems deduced are stripped into the assumptions. The $all\_$ variants only strip in theorems which are not themselves implications.

Forward Chaining (III)

For example:

SML

```
set_goal([], ⌜∀a b c d•a ≤ b ∧ b ≤ c ∧ c ≤ d ⇒ a ≤ d⌝);
a(REPEAT strip_tac);
```

ProofPower Output

```
(∗  3 ∗)  ⌜a ≤ b⌝
(∗  2 ∗)  ⌜b ≤ c⌝
(∗  1 ∗)  ⌜c ≤ d⌝

(∗ ?⊢ ∗)  ⌜a ≤ d⌝
```

SML

```
a(fc_tac[≤_trans_thm]);
```

ProofPower Output

```
(∗  6 ∗)  ⌜a ≤ b⌝
(∗  5 ∗)  ⌜b ≤ c⌝
(∗  4 ∗)  ⌜c ≤ d⌝
(∗  3 ∗)  ⌜∀ n• b ≤ n ⇒ a ≤ n⌝
(∗  2 ∗)  ⌜∀ n• c ≤ n ⇒ b ≤ n⌝
(∗  1 ∗)  ⌜∀ n• d ≤ n ⇒ c ≤ n⌝

(∗ ?⊢ ∗)  ⌜a ≤ d⌝
```

SML

```
a(all_asm_fc_tac[] THEN all_asm_fc_tac[]);
```

ProofPower Output

```
Tactic produced 0 subgoals:
Current and main goal achieved
```


Forward Chaining (IV)

- Many useful properties are naturally formulated as universally quantified implications:

$$\begin{array}{ll} \leq\_trans\_thm & \vdash \forall\ m\ i\ n \bullet\ m \leq i \land i \leq n \Rightarrow m \leq n \\ less\_trans\_thm & \vdash \forall\ m\ i\ n \bullet\ m < i \land i < n \Rightarrow m < n \\ mod\_less\_thm & \vdash \forall\ m\ n \bullet\ 0 < n \Rightarrow m\ Mod\ n < n \end{array}$$

  Forward chaining saves having to specialise such facts explicitly.

- A function, fc_canon, is used to generate implications from the arguments to the forward chaining. E.g.,

$$\begin{array}{l} \vdash (A \land B) \lor C \\ \vdash \forall\ m\ i\ n \bullet\ m \leq i \land i \leq n \Rightarrow m \leq n \end{array}$$

  are treated as:

$$\begin{array}{l} \vdash \neg\ B \Rightarrow \neg\ C \Rightarrow F \\ \vdash \neg\ A \Rightarrow \neg\ C \Rightarrow F \\ \vdash \forall\ n\ i\ m \bullet\ m \leq i \Rightarrow i \leq n \Rightarrow \neg\ m \leq n \Rightarrow F \end{array}$$

- The $\Rightarrow F$ part produced by fc_canon is simplified away when the new theorem is stripped into the assumptions.

- The new theorems stripped into the assumptions are made as general as possible by universally quantifying them over any free variables which do not appear in the goal.

Exercises 11 : Forward Chaining

1. Experiment with the various $all\_$ and $asm\_$ variants of fc_tac to prove the following goals:

   (a) (using $\leq\_trans\_thm$)
   $\forall a\ b\ c\ d \bullet a \leq b \wedge b \leq c \wedge c \leq d \Rightarrow a \leq d$

   (b) (no theorem required)
   $\forall X\ Y\ Z \bullet X \subseteq Y \wedge Y \subseteq Z \Rightarrow X \subseteq Z$

   In each case, what is the minimum number of applications of a forward chaining tactic required and why?

2. Can you use forward chaining to simplify the proof of the following example from exercises 10:
   $(\forall x\ y \bullet f\ x \leq f\ y \Rightarrow\ x \leq y) \Rightarrow (\forall x\ y \bullet f\ x = f\ y \Rightarrow x \leq y)$

Exercises 11 : Solutions

SML

$set\_goal([], \ulcorner \forall a\ b\ c\ d \bullet a \le b \land b \le c \land c \le d \Rightarrow a \le d \urcorner);$

$(*\ 1(a)\ *)$

$a(REPEAT\ strip\_tac);$

$a(all\_fc\_tac[\le\_trans\_thm]\ THEN\ all\_fc\_tac[\le\_trans\_thm]);$

$pop\_thm();$

SML

$set\_goal([], \ulcorner \forall X\ Y\ Z \bullet X \subseteq Y \land Y \subseteq Z \Rightarrow X \subseteq Z \urcorner);$ $(*\ 1(b)\ *)$

$a(REPEAT\ strip\_tac);$

$a(all\_asm\_fc\_tac[]\ THEN\ all\_asm\_fc\_tac[]);$

$pop\_thm();$

In both cases, at least 2 applications of forward chaining are needed since a result from one forward chaining pass must be added to the assumptions to "seed" the second pass.

SML

$set\_goal([],$ $(*\ 2\ *)$

$\ulcorner (\forall x\ y \bullet f\ x \le f\ y \Rightarrow x \le y) \Rightarrow (\forall x\ y \bullet f\ x = f\ y \Rightarrow x \le y) \urcorner);$

$a(REPEAT\ strip\_tac);$

$a(lemma\_tac \ulcorner f\ x \le f\ y \urcorner\ THEN1\ asm\_rewrite\_tac[]);$

$a(all\_asm\_fc\_tac[]);$

$pop\_thm();$

Proof Contexts

- A **proof context** is a named collection of settings of parameters for many of the tactics, conversions, rules etc.

- Customises many parts of the system including:
  - stripping (*strip_tac*, *strip_asm_tac* etc.)
  - rewriting (*rewrite_tac* etc.)
  - automatic proof (*prove_tac*, *asm_prove_tac*)
  - automatic existence proof (*prove_∃_tac*)

- Some proof contexts recommended for everyday use:

predicate calculus	*predicates*
sets	*sets_ext1*
above + lists etc.	*hol2*, *hol*

- use **get_pcs** to list the proof context names together with the theory each proof context belongs to.

Names with ′ are *component* proof contexts: mainly intended for use in conjunction with others.

Names without ′ are *complete* proof contexts: usable on their own.

Using Proof Contexts

- Switch proof context for just one tactic, conversion or rule using:

SML
```
PC_T : string -> TACTIC -> TACTIC;
PC_T1 : string -> ('a -> TACTIC) -> 'a -> TACTIC;

PC_C : string -> CONV -> CONV;
PC_C1 : string -> ('a -> CONV) -> 'a -> CONV;

pc_rule : string -> ('a -> THM) -> 'a -> THM;
pc_rule1 : string -> ('a -> 'b -> THM) ->
                                 'a -> 'b -> THM;
```

- Work with a proof context over several steps using:

SML
```
set_pc : string -> unit;

push_pc : string -> unit;
pop_pc : unit -> unit;
```

- Work with multiple merged proof contexts using, e.g:

SML
```
MERGE_PCS_T : string list -> TACTIC -> TACTIC;
set_merge_pcs : string list -> unit;
```

etc.

- Find out what proof context is in force using:

SML
```
print_status : unit -> unit;
```


What's in the proof contexts?

SML

PC_C1 "$sets\_ext1$" $rewrite\_conv$[]
$\ulcorner\{(1,\ 2)\} \subseteq \{(x,\ y) \mid x + 1 \leq y\} \lor 4 > 5\urcorner$;

ProofPower Output:

$val\ it = \vdash \{(1,\ 2)\} \subseteq \{(x,\ y)|x + 1 \leq y\} \lor 4 > 5$
$\Leftrightarrow (\forall\ x1\ x2\bullet\ (x1,\ x2) = (1,\ 2) \Rightarrow x1 + 1 \leq x2)$
$\lor 4 > 5 : THM$

SML

PC_C1 "$hol2$" $rewrite\_conv$[]
$\ulcorner\{(1,\ 2)\} \subseteq \{(x,\ y) \mid x + 1 \leq y\} \lor 4 > 5\urcorner$;

ProofPower Output:

$val\ it = \vdash \{(1,\ 2)\} \subseteq \{(x,\ y)|x + 1 \leq y\} \lor 4 > 5$
$\Leftrightarrow (\forall\ x1\ x2\bullet\ x1 = 1 \land x2 = 2 \Rightarrow x1 + 1 \leq x2) : THM$

SML

PC_C1 "$hol2$" $rewrite\_conv$[]$\ulcorner A \cap A \subseteq B\urcorner$;

ProofPower Output:

$val\ it = \vdash A \cap A \subseteq B \Leftrightarrow (\forall\ x\bullet\ x \in A \Rightarrow x \in B) : THM$

SML

PC_C1 "hol" $rewrite\_conv$[]$\ulcorner A \cap A \subseteq B\urcorner$;

ProofPower Output:

$val\ it = \vdash A \cap A \subseteq B \Leftrightarrow A \subseteq B : THM$

Automatic Proof Procedures

- Proof context component accessed via:

prove_tac	when the conclusion of a goal is automatically provable on its own
asm_prove_tac	when the goal is automatically provable using the assumptions
prove_rule	to state and prove a conjecture automatically

- If you merge several proof contexts, the "*prove_tac*" comes from the last one in the list.

- Many proof contexts contain *basic_prove_tac*. It uses rewriting, a simple heuristic for eliminating equations involving variables, and a few steps of first-order resolution.

  As seen with the theorems from PM and ZRM, this is useful for simple predicate calculus theorems and for elementary facts about sets. For example:

SML

$$prove\_rule[]\ulcorner(\exists x\bullet\ \phi x) \lor (\exists y\bullet\ \psi y) \Leftrightarrow (\exists z\bullet\ \phi z \lor \psi z)\urcorner;$$
$$prove\_rule[]\ulcorner\forall a\ \ b\bullet a \subseteq b \land b \subseteq a \Leftrightarrow a = b\urcorner;$$

ProofPower Output

$$val\ it = \vdash (\exists\ x\bullet\ \phi\ x) \lor (\exists\ y\bullet\ \psi\ y)$$
$$\Leftrightarrow (\exists\ z\bullet\ \phi\ z \lor \psi\ z) : THM$$
$$val\ it = \vdash \forall\ a\ b\bullet\ a \subseteq b \land b \subseteq a \Leftrightarrow a = b : THM$$

Linear Arithmetic (I)

- Proof context **lin_arith** contains an automatic proof procedure for **linear arithmetic**.

- Useful for many simple arithmetic problems. For example,

SML

$$\textit{pc\_rule1}\ \texttt{"}\textit{lin\_arith}\texttt{"}\ \textit{prove\_rule}[]\ \ulcorner a \le b \land a + b < c + a \Rightarrow a < c \urcorner;$$

ProofPower Output

$$\textit{val it} = \vdash a \le b \land a + b < c + a \Rightarrow a < c : \textit{THM}$$

- Strictly speaking, "linear arithmetic" means terms built up from:

  "Atoms" (numeric literals, variables of type $\mathbb{N}$, etc.)
  Multiplication by numeric literals
  Addition
  $=, \le, \ge, <, >$
  Logical operators

- E.g. all the following are terms of linear arithmetic:

$$\forall a\ c \bullet (\exists b \bullet a \ge b \land \neg\ b < c) \Rightarrow a \ge c$$

$$\forall a\ b\ c \bullet a + 2*b < 2*a \Rightarrow b + b < a$$

$$\forall\ x\ y \bullet \neg\ (2*x + y = 4 \land 4*x + 2*y = 7)$$

Linear Arithmetic (II)

- Rewriting/stripping in lin_arith processes numeric relations by "multiplying out and collecting like terms".

SML

```
pc_rule1 "lin_arith" rewrite_conv[]
        ⌜(i + j)*(j + i) ≤ j*j + j⌝;
```

ProofPower Output

```
val it = ⊢ (i + j) * (j + i) ≤ j * j + j
        ⇔ i * i + 2 * i * j ≤ j : THM
```

$i * i$, $i * j$ and j now treated as atoms.

So a little more general than "strict" linear arithmetic.

- $\neg(a < 1 + 2{*}b \wedge 4{*}b < 2{*}a)$ is proved thus:

if	(1)	$a \leq 2 * b$
and	(2)	$4 * b + 1 \leq 2 * a$
then	2*(1) +(2)	$2 * a + 4 * b + 1 \leq 2 * a + 4 * b$
whence		$1 \leq 0$
whence		CONTRADICTION

Exercises 12: Proof Contexts

1. Using $REPEAT\ strip\_tac$ and $asm\_rewrite\_tac$ prove

   $$(\forall x\ y \bullet f(x,\ y) = (y,\ x)) \Rightarrow \forall x\ y \bullet f(f\ (x,\ y)) = (x,\ y)$$

   Apply the tactics one at a time rather than using $THEN$. Now set the proof context to "$predicates$" using set_pc and prove it again. What differences do you observe?

   Set the proof context back to "$hol2$" when you've finished.

2. Prove the following

   (a) $\{(x,\ y) \mid \neg x = 0 \land y = 2*x\} \subseteq \{(x,\ y) \mid x < y\}$

   (b) $\{(x,\ y) \mid x \geq 2 \land y = 2*x\} \subseteq \{(x,\ y) \mid x + 1 < y\}$

   (c) $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$

   (d) $\forall m \bullet \{i \mid m \leq i \land i < m + 3\} = \{m;\ m+1;\ m+2\}$

   (e) $\{i \mid 5*i = 6*i\} = \{0\}$

Exercises 12: Solutions

SML

```
                                                              (* 1 *)
set_goal([], ⌜(∀x y•f(x, y) = (y, x)) ⇒ ∀x y•f(f (x, y)) = (x, y)⌝);
a(REPEAT strip_tac);
(* *** Goal "1" *** *)
a(asm_rewrite_tac[]);
(* *** Goal "2" *** *)
a(asm_rewrite_tac[]);
pop_thm();
```

SML

```
set_pc"predicates";
set_goal([], ⌜(∀x y•f(x, y) = (y, x)) ⇒ ∀x y•f(f (x, y)) = (x, y)⌝);
a(REPEAT strip_tac);
a(asm_rewrite_tac[]);
pop_thm();
set_pc"hol2";
```

The second proof is shorter because the proof context *predicates* does not cause equations between pairs to be split into pairs of equations.

SML

```
                                                              (* 2 *)
map (merge_pcs_rule1["hol2", "lin_arith"] prove_rule[]) [
(* (a) *)   ⌜{(x, y) | ¬x = 0 ∧ y = 2*x} ⊆ {(x, y) | x < y}⌝,
(* (b) *)   ⌜{(x, y) | x ≥ 2 ∧ y = 2*x} ⊆ {(x, y) | x + 1 < y}⌝,
(* (d) *)   ⌜∀m•{i | m ≤ i ∧ i < m + 3} = {m; m+1; m+2}⌝,
(* (e) *)   ⌜{i | 5*i = 6*i} = {0}⌝];
(* (c) *)   pc_rule1 "sets_ext1" prove_rule[]
                        ⌜A ∪ (B ∩ C) = (A ∪ B) ∩ (A ∪ C)⌝;
```

(Alternatively, use the subgoal package and *PC_T1*.).

Case Study: Vending Machine System Model

The following paragraphs give a model of a simple vending machine:

SML

new_theory"vm";

HOL Labelled Product

VM_State

$$\begin{array}{ll} takings & : \mathbb{N}; \\ stock & : \mathbb{N}; \\ price & : \mathbb{N}; \\ cash\_tendered & : \mathbb{N} \end{array}$$

HOL Constant

$$vm : VM\_State \to VM\_State$$

$$\begin{array}{lll}
\forall st \bullet & vm\ st & \\
= & if & stock\ st = 0 \\
 & then & MkVM_State \\
 & & (takings\ st)\ (stock\ st)\ (price\ st)\ 0 \\
 & else & if\ cash_tendered\ st < price\ st \\
 & then & st \\
 & else & if\ cash_tendered\ st = price\ st \\
 & then & MkVM_State \\
 & & (takings\ st + cash_tendered\ st) \\
 & & (stock\ st - 1)\ (price\ st)\ 0 \\
 & else & MkVM_State \\
 & & (takings\ st)\ (stock\ st)\ (price\ st)\ 0
\end{array}$$

Case Study: Vending Machine Discussion (I)

- the state of the vending machine is defined as a labelled record type VM_State.

- labelled record type declares **projection functions**:

Projection Functions

$$
\begin{array}{|ll}
takings: & VM\_State \to \mathbb{N} \\
stock: & VM\_State \to \mathbb{N} \\
price: & VM\_State \to \mathbb{N} \\
cash\_tendered: & VM\_State \to \mathbb{N}
\end{array}
$$

If st is a state value, $takings\ st$ is like $st.takings$ in Z or Pascal or Ada.

- also introduces **constructor functions**:

Constructor Function

$$
\begin{array}{|ll}
MkVM\_State: & \mathbb{N} \to \mathbb{N} \to \mathbb{N} \to \mathbb{N} \to VM\_State
\end{array}
$$

If t, s, p, and ct are numbers, $MkVM\_State\ t\ s\ p\ ct$ is a state value with those numbers as its components.

Case Study: Vending Machine Discussion (II)

- Can test the behaviour of the vending machine model by rewriting.
- E.g. introduce a conversion to do this

SML

$val\ run\_vm = rewrite\_conv[get\_spec\ulcorner vm\urcorner,\ get\_spec\ulcorner MkVM\_State\urcorner];$

ProofPower Output

$val\ run\_vm = fn : CONV$

- Now look at test cases

SML

$run\_vm\ \ulcorner vm\ (MkVM\_State\ 0\ 20\ 5\ 5)\urcorner;$

$run\_vm\ \ulcorner vm\ (MkVM\_State\ t\ 20\ 5\ 5)\urcorner;$

ProofPower Output

$$val\ it = \vdash vm\ (MkVM\_State\ 0\ 20\ 5\ 5) = MkVM\_State\ 5\ 19\ 5\ 0 : THM$$

$$val\ it = \vdash vm\ (MkVM\_State\ t\ 20\ 5\ 5) = MkVM\_State\ (t + 5)\ 19\ 5\ 0 : THM$$

- Second test case does **symbolic** execution

Case Study: Vending Machine Critical Requirements

Informal statement of critical requirement: *"No transaction of the vending machine causes the machine's owner to lose money"*.

We formalise this by specifying the set of transition functions which never reduce the value of the machine's contents. The value of a state is computed by the following function.

HOL Constant

$$value : VM\_State \to \mathbb{N}$$

$$\forall st \bullet value\ st = takings\ st + stock\ st * price\ st$$

The set of machines satisfying the critical requirement is then:

HOL Constant

$$vm\_ok : (VM\_State \to VM\_State)\ SET$$

$$\begin{array}{lll} & vm\_ok & \\ = & \{ & trf \\ & \mid & \forall cb\ s\ p\ ct \bullet \\ & & let \quad s1 = MkVM\_State\ cb\ s\ p\ ct \\ & & in\ let \quad s2 = trf\ s1 \\ & & in \quad value\ s2 \geq value\ s1\} \end{array}$$

Exercises 13: Case Study

First of all execute the new_theory command and the 4 paragraphs of the vending machine specification.

1. Execute the definition of run_vm:

SML

```
val run_vm = rewrite_conv[get_spec⌜vm⌝, get_spec⌜MkVM_State⌝];
```

   Experiment with the model by using run_vm to see what it does on various test data. What does the vending machine do if the price is set to 0?

2. Prove that the model of the vending machine satisfies its critical requirements. I.e., prove:

$$\mathbf{vm} \in \mathbf{vm\_ok}$$

   Hints:

   (a) Try $REPEAT\ strip\_tac$

   (b) Try rewriting with the definitions of any of $MkVM\_State$, vm, vm_ok or $worth$ which appear in the goal.

   (c) let-expressions may be eliminated by rewriting with let_def.

   (d) Is there an if-term in the goal? Can you use $\mathbb{N}\_cases\_thm$ or $less\_cases\_thm$ (together with $strip\_asm\_tac$ and $\forall\_elim$ or $list\_\forall\_elim$) to perform the relevant case analysis?

   (e) If you believe the goal is true by dint of arithmetic facts alone try PC_T1"lin_arith" asm_prove_tac[].

   (f) If none of the above hints apply, do you have an if-term which could be simplified using an "obvious" arithmetic consequence of your assumptions. If so set the "obvious" consequence up as a lemma with $lemma\_tac$.

Exercise 13/1: Solution

The following test cases check out each branch of the *if*-terms in the definition of *vm*:

Branch 1: out of stock: the machine refunds any cash tendered.

SML
```
run_vm ⌜vm (MkVM_State t 0 p ct)⌝;
```

Branch 2: in stock; cash tendered is less than the price: the machine waits for more cash to be tendered:

SML
```
run_vm ⌜vm (MkVM_State t 20 5 2)⌝;
```

Branch 3: in stock; cash tendered is equal to the price: the machine dispenses a chocolate bar and adds the cash tendered to its takings:

SML
```
run_vm ⌜vm (MkVM_State t 20 5 5)⌝;
```

Branch 4: in stock; cash tendered exceeds the price: the machine refunds the cash tendered:

SML
```
run_vm ⌜vm (MkVM_State t 20 5 6)⌝;
```

If the price is set to *0*, the machine first refunds any cash tendered and then gives away all the stock!

SML
```
run_vm ⌜vm (MkVM_State t 4 0 6)⌝;
run_vm ⌜vm (MkVM_State t 4 0 0)⌝;
run_vm ⌜vm (MkVM_State t 3 0 0)⌝;
run_vm ⌜vm (MkVM_State t 2 0 0)⌝;
run_vm ⌜vm (MkVM_State t 1 0 0)⌝;
run_vm ⌜vm (MkVM_State t 0 0 0)⌝;
```


Exercise 13/2: Solution

SML

```
set_goal([], ⌜vm ∈ vm_ok⌝);
(* Goal "": Expand definitions and let−terms: *)
a(rewrite_tac [get_spec ⌜vm_ok⌝, get_spec⌜vm⌝,
                              get_spec⌜MkVM_State⌝, let_def]);

(* Goal "": remove outer universal quantifiers *)
a(REPEAT strip_tac);

(* Goal "": case split on the amount of stock:
                                          s = 0 ∨ s = i + 1 for some i *)
a(strip_asm_tac(∀_elim⌜s⌝ ℕ_cases_thm) THEN asm_rewrite_tac[]);

(* Goal "1": s = 0 *)
a(asm_rewrite_tac[get_spec⌜value⌝, get_spec⌜MkVM_State⌝]);

(* Goal "2": case split on ct < p: ct < p ∨ ct = p ∨ p < ct *)
a(strip_asm_tac(list_∀_elim[⌜ct⌝, ⌜p⌝] less_cases_thm));

(* Goal "2.1": ct < p: *)
a(asm_rewrite_tac[get_spec⌜MkVM_State⌝]);

(* Goal "2.2": ct = p: *)
a(asm_rewrite_tac[get_spec⌜value⌝, get_spec⌜MkVM_State⌝]);
a(PC_T1 "lin_arith" asm_prove_tac[]);

(* Goal "2.3": ct > p: need ¬ct < p ∧ ¬ ct = p to evaluate if *)
a(lemma_tac ⌜¬ct < p ∧ ¬ ct = p⌝ THEN1
            PC_T1 "lin_arith" asm_prove_tac[]);
a(asm_rewrite_tac[get_spec⌜value⌝, get_spec⌜MkVM_State⌝]);

val vm_ok_thm = pop_thm();
```


Proof Strategy

- A large application proof may take several man years of effort to complete.

- Top level proof strategy for large proofs must be carefully thought out.

  The lemmas are best proven separately, stored in the theory, and combined in a top level proof delivering the required result from the major lemmas. Exploration may be forwards or backwards.

- Lemmas of moderate size may be proven using the goal package.

  Such a proof would consist of a combination of stripping, rewriting with definitions, assumptions and previously proven results, and other uses of previous results.

What to do when faced with a Goal
Sanity Checks

- Decide whether the goal is true, if not, don't try to prove it!

- Decide whether the conclusion is relevant (are the assumptions inconsistent?).

- Do you see what the goal means? If not, can you simplify it.

- If all else fails, try retracing your steps.

What to do when faced with a Goal
Main Choices

- Decompose by stripping or contradiction ($strip\_tac$, $contr\_tac$)
- Work forwards from assumptions (e.g. $spec\_asm\_tac$, fc_tac)
- Do a case split ($strip\_asm\_tac$, $cases\_tac$)
- Swap the conclusion with an assumption ($swap\_asm\_concl\_tac$)
- Prove a lemma ($lemma\_tac$)
- Prove automatically (e.g. asm_prove_tac, $prove\_\exists\_tac$)
- Transform the conclusion by rewriting (e.g. with a definition)
- Induction ($\ldots\_induction\_tac$)

Exercises 14.

1. Use *contr_tac*, and *spec_asm_tac* and rewriting prove that there is no greatest natural number:

SML

$set\_goal([], \ulcorner \forall m \bullet \exists n \bullet\ m < n \urcorner);$

(Hint: $m < m + 1$).

2. Rather than using *contr_tac*, it is often more natural to prove goals with existentially quantified conclusions directly. $\exists\_tac$ lets you do this by supplying a term to act as a "witness". Use $\exists\_tac$ to give a more natural solution to the previous exercise:

SML

$set\_goal([], \ulcorner \forall m \bullet \exists n \bullet\ m < n \urcorner);$

3. Prove that there is no onto function from the natural numbers to the set of all numeric functions on the natural numbers:

SML

$set\_goal([], \ulcorner \forall f : \mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{N}) \bullet \exists g \bullet \forall i \bullet \neg f\ \ i = g \urcorner);$

(Hints: Note that for f of the above type, $\lambda j \bullet (f\ j\ j) + 1$ cannot be in the range of f. Rewriting with *ext_thm* is useful for reasoning about equations between functions.)

4. It can happen that an equation is the wrong way round for use as a rewrite rule. The usual means for dealing with this type of problem is the conversion eq_sym_conv. Like other conversions this may be propagated over a term using the conversionals MAP_C and $ONCE\_MAP\_C$. Execute the following lines one at a time to see what happens:

```
eq_sym_conv ⌜1 + 1 + 1 = 3⌝;
eq_sym_conv ⌜∀x•x + x + x = 3*x⌝;
ONCE_MAP_C eq_sym_conv ⌜∀x•x + x + x = 3*x⌝;
```

A conversion may be converted into a tactic using $conv\_tac$. Use this and the conversion and conversional you have just experimented with together with the tactics $swap\_asm\_concl\_tac$ and the theorems ext_thm and $comb\_k\_def$ to prove the following:

SML
```
set_goal([], ⌜∀f:'a→'b→'a•(∀x y•x = f x y) ⇒ f = CombK⌝);
```

(Hint: take care to avoid looping rewrites by using the "once" rewriting tactics while you look for the proof.)

5. A common way of using a theorem is to to strip it into the assumptions. This is done with $strip\_asm\_tac$. Very often one specialises the theorem with $\forall\_elim$ or $list\_\forall\_elim$ before stripping it in and sometimes one may wish to use $rewrite\_rule$ to rewrite it too. Use the theorem $div\_mod\_unique\_thm$ in this way to prove:

SML
```
set_goal([], ⌜∀i j•0 < i ⇒ (i * j) Div i = j⌝);
```

(Hints: rewrite the theorem with $times\_comm\_thm$ suitably specialised to identify subterms of the form $i * j$ and $j * i$ into the same form; use the technique of the previous exercise to avoid a looping rewrite with the assumption added by $strip\_asm\_tac$).

6. Execute the following paragraph to define a function σ which maps i to the sum of the first i positive integers:

HOL Constant

$$\sigma : \mathbb{N} \to \mathbb{N}$$

$$\sigma\ 0 = 0$$
$$\land \quad \forall i \bullet\ \sigma(i{+}1) = \sigma\ i + (i + 1)$$

The consistency of this paragraph should be proved automatically. Check this by using *get_spec* to get the defining axiom for σ, which should have no assumptions. Prove the following theorem:

SML

```
set_goal([], ⌜∀i•σ i = (i*(i + 1)) Div 2⌝);
```

(Hint: use induction to prove a lemma that $i * (i + 1) = 2 * \sigma i$ and then use the result of the previous exercise; the lemma may be proved by rewriting with assumptions and the definition of σ and then using the proof context *lin_arith*.)

7. Construct a paragraph defining a function ϕ such that for positive i, ϕi is the i^{th} element of the Fibonacci sequence, $1, 1, 2, 3, 5, \ldots$, where each number is the sum of the previous two. Does the system automatically prove the consistency of your definition?

8. If you did the previous exercise, delete the function ϕ you defined (using $delete\_const$). Enter the following paragraphs which define ϕ using an auxiliary function γ:

HOL Constant

$$\gamma : \mathbb{N} \to (\mathbb{N} \times \mathbb{N})$$

$$\gamma\ 0 = (0,\ 1)$$
$$\land \quad \forall i \bullet \gamma(i+1) = let\ (a,\ b) = \gamma\ i\ in\ (b,\ a + b)$$

HOL Constant

$$\phi : \mathbb{N} \to \mathbb{N}$$

$$\forall i \bullet \phi\ i = Fst\ (\gamma\ i)$$

These definitions are proved consistent automatically. Prove that ϕ does indeed compute the Fibonacci numbers:

```
set_goal([], ⌜
        φ 0 = 0
∧       φ 1 = 1
∧       ∀i•φ(i+2) = φ(i+1) + φ i
⌝);
```

(Hints: first rewrite with the definition of ϕ; then prove a lemma or lemmas showing how $\gamma\ 1$ and $\gamma(i + 2)$ may be rewritten so that the definition of γ may be used to rewrite them.)

9. The approach of the previous exercise has the disadvantage that the specification was not as abstract as one might like. A cleaner approach is to use the obvious definition of ϕ, and then prove that it is consistent using a function γ which is only introduced as a variable during the course of the proof. The tactic $prove\_\exists\_tac$ gives access to the mechanisms that the system uses in its attempt to prove that paragraphs are consistent.

We demonstrate the above technique in this exercise.

9.(cont) First of all, delete the function γ that you defined in the previous exercise (using $delete\_const$, which will also cause ϕ to be deleted).

SML

$delete\_const \ulcorner \gamma \urcorner$;

Enter the following paragraph which gives the natural definition of ϕ:

HOL Constant

$$\phi : \mathbb{N} \to \mathbb{N}$$

$$\phi\ 0 = 0$$
$$\land \quad \phi\ 1 = 1$$
$$\land \quad \forall i \bullet \phi(i+2) = \phi(i+1) + \phi\ i$$

Examine the theorem that get_spec returns for ϕ, it has a consistency caveat as an assumption. Discharge this consistency caveat as follows:

First of all go into the subgoaling package using the following command:

$push\_consistency\_goal \ulcorner \phi \urcorner$;

Now set as a lemma the existence of a γ as in the previous exercise; the lemma is proved immediately by $prove\_\exists\_tac$ and you can then use $\exists\_tac \ulcorner \lambda i \bullet Fst(\gamma\ i) \urcorner$ followed a proof almost identical with the previous exercise (hint: $rewrite\_tac$ will eliminate the β-redexes introduced when you apply $\exists\_tac$). Save the consistency theorem using the following command:

$save\_consistency\_thm \ulcorner \phi \urcorner (pop\_thm())$;

If you now examine the theorem that get_spec returns for ϕ, you should see that it no longer has an assumption.

(Note: the variable name 'ϕ'', created by decorating 'ϕ' is displayed by the pretty printer as \$ "ϕ'" since it violates the HOL lexical rules for identifiers. The parser will accept identifiers violating the normal lexical rules if they are presented in this way.)

Exercises 14: Solutions

SML

```
(* no. 1 *)
set_goal([], ⌜∀m•∃n• m < n⌝);
a(contr_tac);
a(spec_asm_tac⌜∀ n• ¬ m < n⌝⌜m+1⌝);
val thm1 = pop_thm();
```

SML

```
(* no. 2 *)
set_goal([], ⌜∀m•∃n• m < n⌝);
a(REPEAT strip_tac);
a(∃_tac⌜m+1⌝);
a(rewrite_tac[]);
val thm2 = pop_thm();
```

SML

```
(* no. 3 *)
set_goal([], ⌜∀f : ℕ → (ℕ → ℕ)•∃g•∀i•¬f i = g⌝);
a(REPEAT strip_tac);
a(∃_tac⌜λj•(f j j) + 1⌝);
a(rewrite_tac[ext_thm]);
a(REPEAT strip_tac);
a(∃_tac⌜i⌝ THEN REPEAT strip_tac);
val thm3 = pop_thm();
(* no. 4 *)
set_goal([], ⌜∀f:'a→'b→'a•(∀x y•x = f x y) ⇒ f = CombK⌝);
a (REPEAT strip_tac);
a (rewrite_tac[ext_thm, comb_k_def]);
a (swap_asm_concl_tac⌜∀ x y• x = f x y⌝);
a (conv_tac(ONCE_MAP_C eq_sym_conv));
a (swap_asm_concl_tac⌜¬ f x x' = x⌝ THEN asm_rewrite_tac[]);
val thm4 = pop_thm();
```

SML

```
(* no. 5 *)
set_goal([], ⌜∀i j•0 < i ⇒ (i * j) Div i = j⌝);
a (REPEAT strip_tac);
a (strip_asm_tac(
        rewrite_rule[∀_elim⌜j⌝times_comm_thm]
        (list_∀_elim[⌜i*j⌝, ⌜i⌝, ⌜j⌝, ⌜0⌝] div_mod_unique_thm)));
a (swap_asm_concl_tac⌜j = (i * j) Div i⌝ THEN
                (conv_tac(ONCE_MAP_C eq_sym_conv)));
a (strip_tac);
val thm5 = pop_thm();
```

SML

```
(* no. 6 *)
set_goal([], ⌜∀i•σ i = (i*(i + 1)) Div 2⌝);
a (REPEAT strip_tac);
a (lemma_tac⌜i * (i + 1) = 2 * σ i⌝);
(* *** Goal "1" *** *)
a (induction_tac⌜i⌝ THEN asm_rewrite_tac[get_spec⌜σ⌝]);
a(PC_T1 "lin_arith" asm_prove_tac[]);
(* *** Goal "2" *** *)
a (asm_rewrite_tac[rewrite_rule[](list_∀_elim[⌜2⌝, ⌜σ i⌝]thm5)]);
val thm6 = pop_thm();
```


SML

```
(* no. 7 *)
```

The obvious way of defining the Fibonacci function is not automatically proved consistent:

SML

$delete\_const ⌜\phi⌝;$

HOL Constant

$$\phi : \mathbb{N} \to \mathbb{N}$$

$$\phi\ 0 = 0$$
$$\land \quad \phi\ 1 = 1$$
$$\land \quad \forall i \bullet \phi(i+2) = \phi(i+1) + \phi\ i$$

SML

$get\_spec ⌜\phi⌝;$

SML

```
delete_const⌜φ⌝;
(* no. 8 *)
```

HOL Constant

$$\gamma : \mathbb{N} \to (\mathbb{N} \times \mathbb{N})$$

$$\gamma\ 0 = (0,\ 1)$$
$$\land \quad \forall i \bullet \gamma(i+1) = let\ (a,\ b) = \gamma\ i\ in\ (b,\ a + b)$$

HOL Constant

$$\phi : \mathbb{N} \to \mathbb{N}$$

$$\forall i \bullet \phi\ i = Fst\ (\gamma\ i)$$

SML

```
set_goal([], ⌜
        φ 0 = 0
∧       φ 1 = 1
∧       ∀i•φ(i+2) = φ(i+1) + φ i
⌝);
a (rewrite_tac[get_spec⌜φ⌝]);
a (lemma_tac⌜γ 1 = γ(0 + 1) ∧ ∀i• γ(i + 2) = γ((i+1)+1)⌝);
(* *** Goal "1" *** *)
a (rewrite_tac[plus_assoc_thm]);
(* *** Goal "2" *** *)
a (pure_asm_rewrite_tac[get_spec⌜γ⌝, let_def] THEN rewrite_tac[]);
val thm8 = pop_thm();
```


SML

```
(* no. 9 *)
delete_const⌜γ⌝;
```

HOL Constant

$$\phi : \mathbb{N} \to \mathbb{N}$$

$$\phi\ 0 = 0$$
$$\land \quad \phi\ 1 = 1$$
$$\land \quad \forall i \bullet \phi(i+2) = \phi(i+1) + \phi\ i$$

SML

```
get_spec⌜φ⌝;
push_consistency_goal⌜φ⌝;
a (lemma_tac⌜∃γ•
        γ 0 = (0, 1)
∧       ∀i•γ(i+1) = let (a, b) = γ i in (b, a + b)
⌝);
(* *** Goal "1" *** *)
a (prove_∃_tac);
(* *** Goal "2" *** *)
a (∃_tac⌜λi•Fst(γ i)⌝);
a (rewrite_tac[]);
a (lemma_tac⌜γ 1 = γ(0 + 1) ∧ ∀i• γ(i + 2) = γ((i+1)+1)⌝);
(* *** Goal "2.1" *** *)
a (rewrite_tac[plus_assoc_thm]);
(* *** Goal "2.2" *** *)
a (pure_asm_rewrite_tac[let_def] THEN asm_rewrite_tac[]);
save_consistency_thm ⌜φ⌝ (pop_thm());
get_spec⌜φ⌝;
```

