

Isabelle Tutorial

Christian Sternagel Makarius Wenzel

1 August 2015

CADE-25

History



1969 **Dana Scott:**
Logic for Computable Functions

1969

1993



1969 **Dana Scott:**
Logic for Computable Functions

1969

1993

1972

Robin Milner:
Stanford LCF





1969 **Dana Scott:**
Logic for Computable Functions

1973 **ML:**
Meta Language

1969

1993

1972 **Robin Milner:**
Stanford LCF





1969 **Dana Scott:**
Logic for Computable Functions

1973 **ML:**
Meta Language

1969

1993

1972 **Robin Milner:**
Stanford LCF



1977 **Edinburgh LCF**



1969 **Dana Scott:**
Logic for Computable Functions

1973 **ML:**
Meta Language



1985 **Larry Paulson:**
Cambridge LCF

1969

1993

1972 **Robin Milner:**
Stanford LCF



1977 **Edinburgh LCF**



1969 **Dana Scott:**
Logic for Computable Functions

1973 **ML:**
Meta Language



1985 **Larry Paulson:**
Cambridge LCF

1969

1993

1972 **Robin Milner:**
Stanford LCF



1977 **Edinburgh LCF**

1986 **Isabelle**



1969 **Dana Scott:**
Logic for Computable Functions

1973 **ML:**
Meta Language



1985 **Larry Paulson:**
Cambridge LCF



1993 **Tobias Nipkow:**
Isabelle/HOL

1969

1993

1972

Robin Milner:
Stanford LCF



1977 **Edinburgh LCF**

1986 **Isabelle**

TTY interaction (1979)



(Wikipedia: K. Thompson and D. Ritchie at PDP-11)

Proof General / Emacs interaction (1999)

```

emacs: Group.thy
File Edit Apps Options Buffers Tools Proof-General X-Symbol Help
State Context Goal Attract Undo Next Disc Goto Restart Q.E.D. Find Command Stop Info Help

finally; show ?thesis; .;
qed;

text {*
  With \name{group-right-inverse} already available,
  \name{group-right-unit}\label{thm:group-right-unit} is now
  established much easier.
*};

theorem group_right_unit: "x • one = (x::'a::group)";
proof -;
  have "x • one = x • (inv x • x)";
  by (simp only: group_left_inverse);
  also have "... = x • inv x • x";
  by (simp only: group_assoc);
  also have "... = one • x";
  by (simp only: group_right_inverse);
  also have "... = x";
  by (simp only: group_left_unit);
  finally; show ?thesis; .;
qed;

text {*
  \medskip The calculational proof style above follows typical
  presentations given in any introductory course on algebra. The basic
  technique is to form a transitive chain of equations, which in turn
  are established by simplifying with appropriate rules. The low-level
  logical details of equational reasoning are left implicit.
-----XEmacs: Group.thy (Isabelle/Isar script XS:isar Font Scripting)-----29%
Proof(prove): step 8, depth 1

goal (have):
x • inv x • x = one • x
1. x • inv x • x = one • x

-----XEmacs: *isabelle-goals* (Isabelle/Isar proofstate)-----All-----

```

Document-oriented Prover IDE: Isabelle/jEdit (2015)

The screenshot displays the Isabelle/jEdit IDE interface. The main window shows a theory file named 'Seq.thy' with the following content:

```

section <Finite sequences>

theory Seq
imports Main
begin

datatype 'a seq = Empty | Seq 'a "'a seq"

fun conc :: "'a seq ⇒ 'a seq ⇒ 'a seq"
where
  "conc Empty ys = ys"
| "conc (Seq x xs) ys = Seq x (conc xs ys)"

fun reverse
where
  "reverse Empty = Empty"
| "reverse (Seq x xs) = conc (reverse xs) (Seq x Empty)"

lemma conc_empty: "conc xs Empty = xs"
  by (induct xs) simp_all
  
```

A tooltip is visible over the `reverse` function definition, showing a constant definition:

```

constant "Seq.seq.Seq"
:: 'a ⇒ 'a seq ⇒ 'a seq
  
```

The right-hand pane shows a search filter and a list of results for 'Seq.thy', including the same theory and function definitions shown in the main window.

At the bottom of the IDE, there is an 'Auto update' checkbox (checked), an 'Update' button, a search field, and a zoom level dropdown set to '100%'. Below this, a 'constants' section is visible, showing the `conc` function definition and a 'Found termination order' message: `"(λp. size (fst p)) <*> {}"`.

The status bar at the bottom indicates the file path '13.39 (200/789)' and system information '(isabelle,isabelle,UTF-8-Isabelle)Nimrod UG 55/410MB 11:45 PM'.

Orchestration of (dis)provers in Isabelle/jEdit

```

theory Scratch
imports Main
begin

datatype 'a tree = Tip | Tree 'a "'a tree" "'a tree"

fun tree_of_list :: "'a list ⇒ 'a tree"
where
  "tree_of_list [] = Tip"
| "tree_of_list (x # xs) = Tree x Tip (tree_of_list xs)"

fun list_of_tree :: "'a tree ⇒ 'a list"
where
  "list_of_tree Tip = []"
| "list_of_tree (Tree x t1 t2) = x # list_of_tree t1 @ list_of_tree t2"

lemma "list_of_tree (tree_of_list xs) = xs"
  by (induct xs) simp_all

lemma "tree_of_list (list_of_tree t) = t"

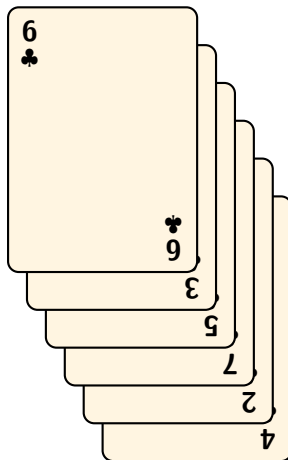
```

Auto Quickcheck found a counterexample:
 $t = \text{Tree } a_1 (\text{Tree } a_1 \text{ Tip Tip}) \text{ Tip}$
 Evaluated terms:
 $\text{tree_of_list (list_of_tree } t) = \text{Tree } a_1 \text{ Tip (Tree } a_1 \text{ Tip Tip)}$

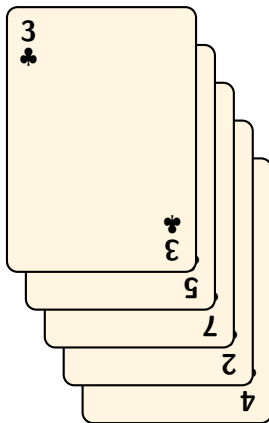
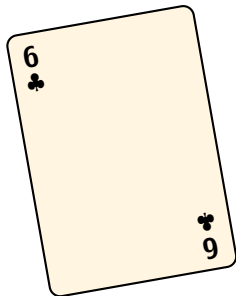
Query
 20,42 (476/477) Input/output complete (isabelle,isabelle,UTF-8-Isabelle)N m r o UG 6/391MB 11:38 PM

Functional programming in Isabelle/HOL

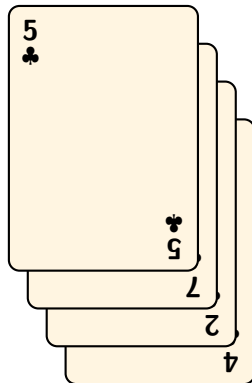
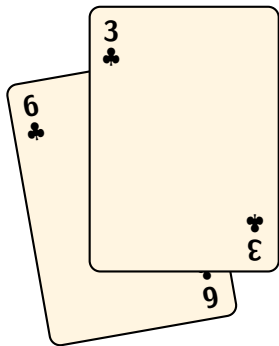
Example: Insertion Sort



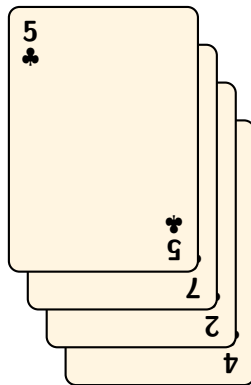
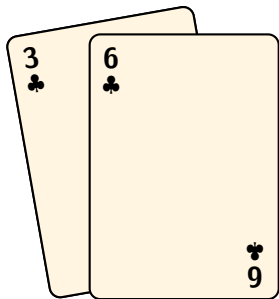
Example: Insertion Sort



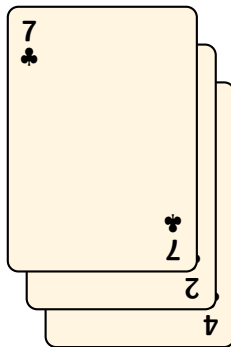
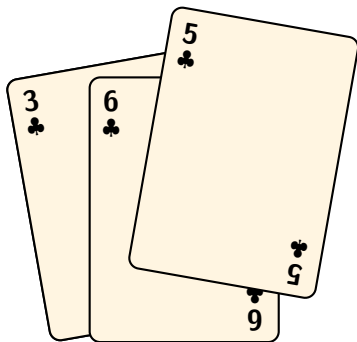
Example: Insertion Sort



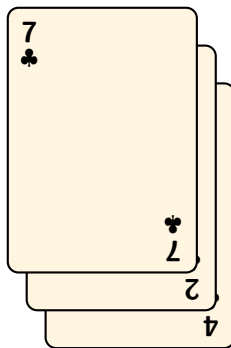
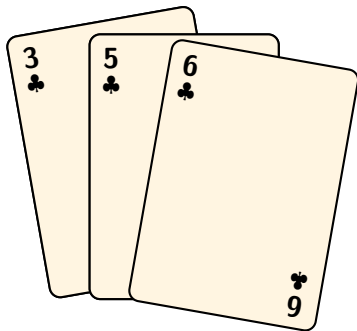
Example: Insertion Sort



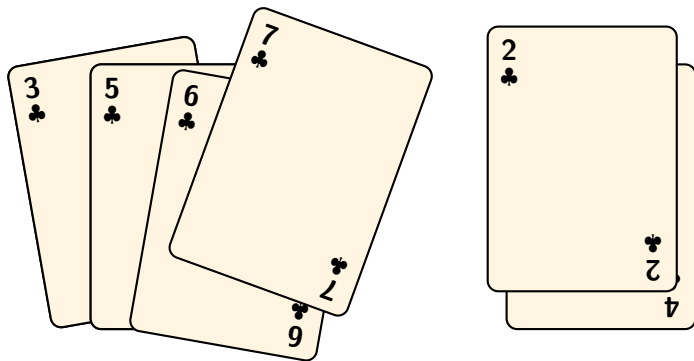
Example: Insertion Sort



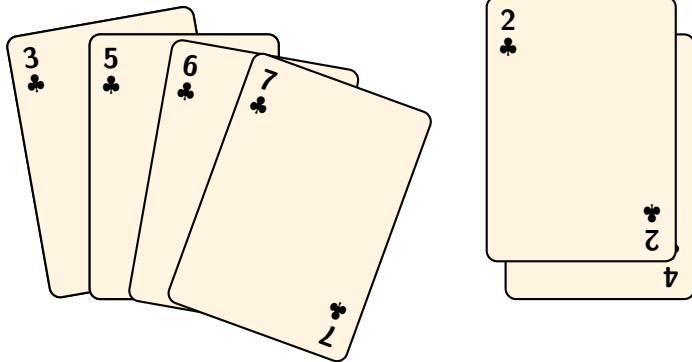
Example: Insertion Sort



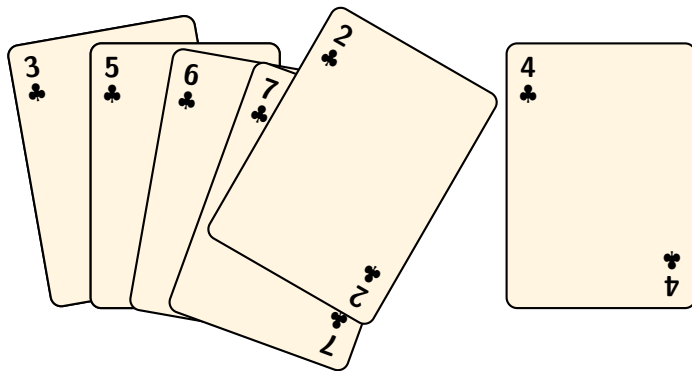
Example: Insertion Sort



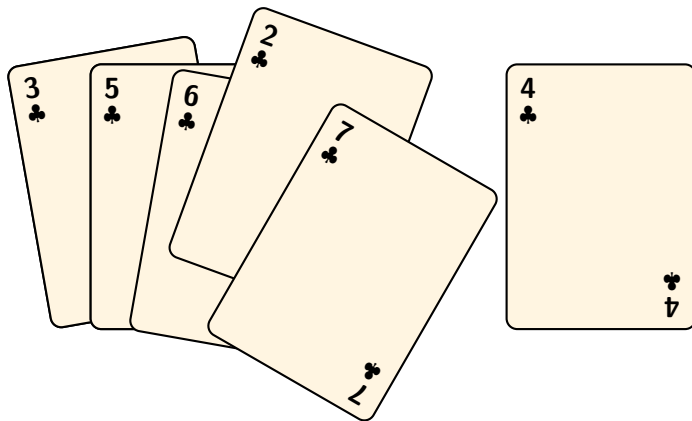
Example: Insertion Sort



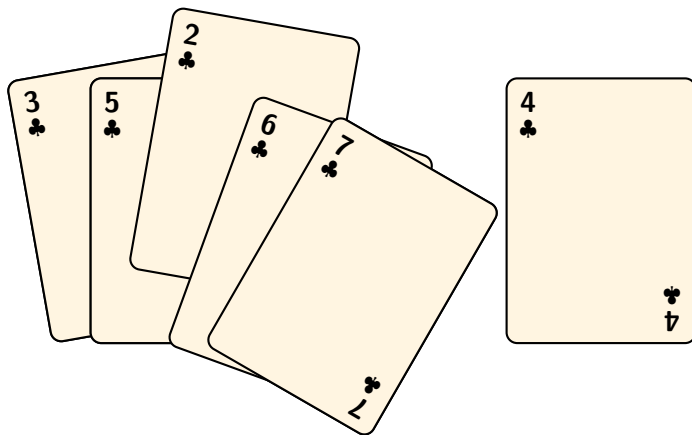
Example: Insertion Sort



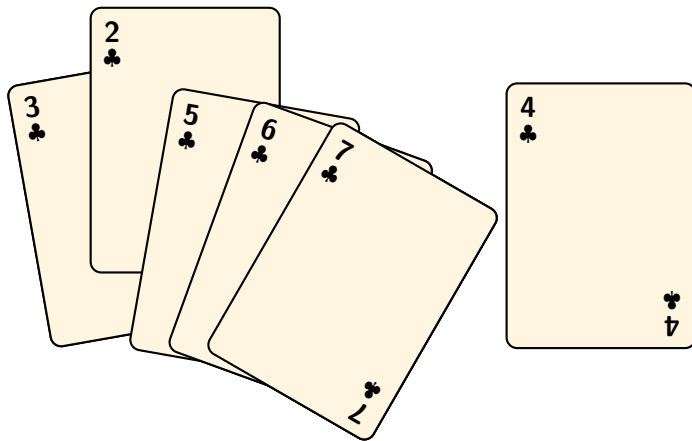
Example: Insertion Sort



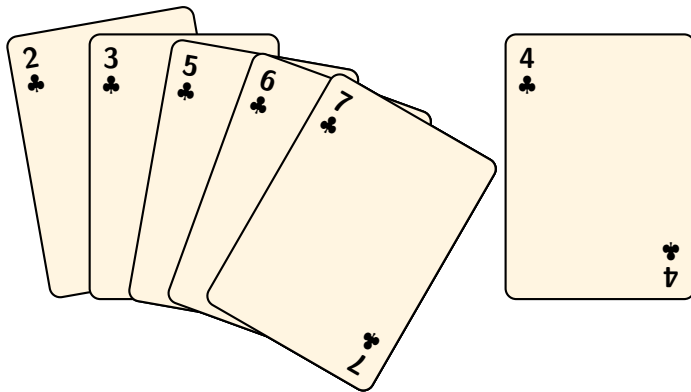
Example: Insertion Sort



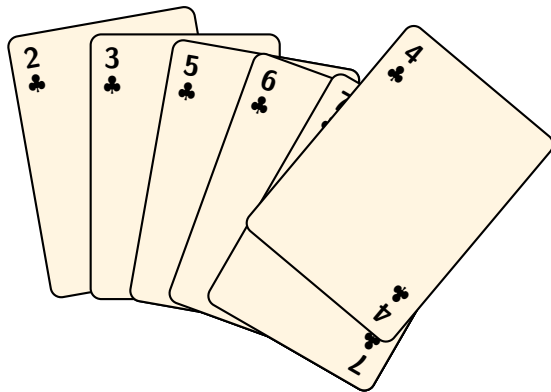
Example: Insertion Sort



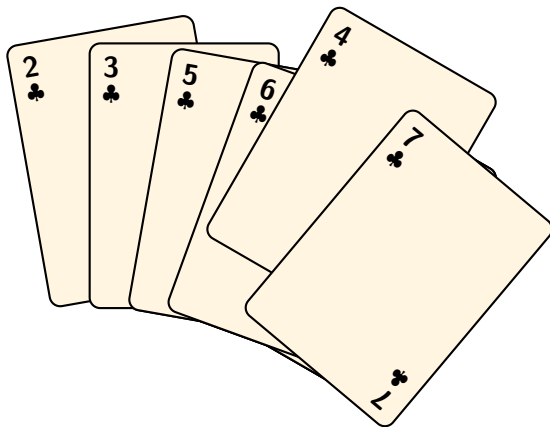
Example: Insertion Sort



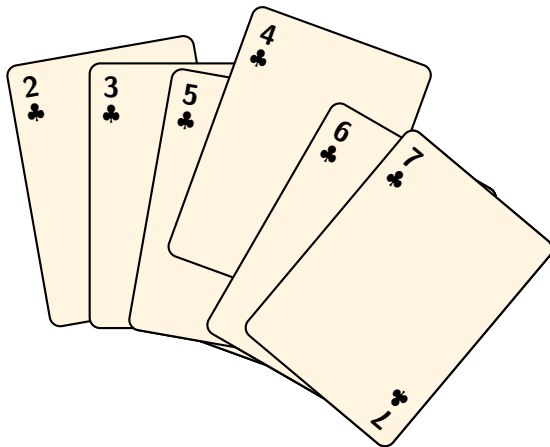
Example: Insertion Sort



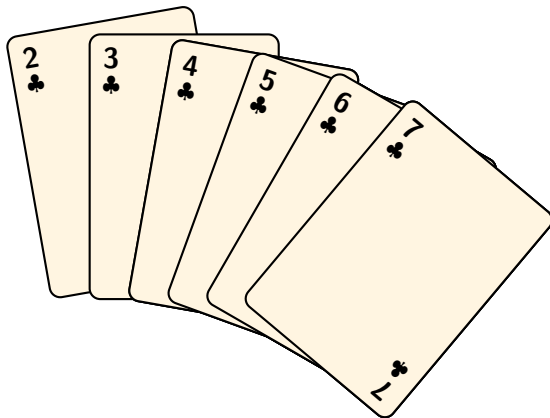
Example: Insertion Sort



Example: Insertion Sort



Example: Insertion Sort



A Functional Implementation

- inserting an element into a sorted list

```
insert :: a -> [a] -> [a]
insert x []      = [x]
insert x (y:ys) = if x ≤ y then x:y:ys
                  else y : insert x ys
```


A Functional Implementation

- inserting an element into a sorted list

```
insert :: a -> [a] -> [a]
insert x []      = [x]
insert x (y:ys) = if x ≤ y then x:y:ys
                  else y : insert x ys
```

- sorting by repeatedly inserting elements into the empty list

```
insertionSort :: [a] -> [a]
insertionSort xs = foldr insert [] xs
```

where

$$\text{foldr } f \ b \ [x_1, x_2, \dots, x_n] = f \ x_1 \ (f \ x_2 \ (\dots (f \ x_n \ b) \dots))$$

A Functional Implementation

- inserting an element into a sorted list

```
insert :: a -> [a] -> [a]
insert x []      = [x]
insert x (y:ys) = if x ≤ y then x:y:ys
                  else y : insert x ys
```

- sorting by repeatedly inserting elements into the empty list

```
insertionSort :: [a] -> [a]
insertionSort xs = foldr insert [] xs
```

where

$$\text{foldr } f \ b \ [x_1, x_2, \dots, x_n] = f \ x_1 \ (f \ x_2 \ (\dots (f \ x_n \ b) \dots))$$

- How to prove correctness?

A Functional Implementation

- inserting an element into a sorted list

```
insert :: a -> [a] -> [a]
insert x []      = [x]
insert x (y:ys) = if x ≤ y then x:y:ys
                  else y : insert x ys
```

- sorting by repeatedly inserting elements into the empty list

```
insertionSort :: [a] -> [a]
insertionSort xs = foldr insert [] xs
```

where

$$\text{foldr } f \ b \ [x_1, x_2, \dots, x_n] = f \ x_1 \ (f \ x_2 \ (\dots (f \ x_n \ b) \dots))$$

- How to prove correctness? \implies Isabelle/HOL

Isabelle/HOL proof tools

Sledgehammer

- heavy external ATPs / SMTs for **proof search**
- light internal ATP (Metis) for **proof reconstruction**

The screenshot shows the Isabelle Sledgehammer interface. The main window displays the following code:

```
theory Scratch
imports Main
begin

lemma "[x] = [y]  $\implies$  x = y" by (metis list.inject)
```

Below the code, the Sledgehammer interface shows the provers used for the proof:

Provers: cvc4 remote_vampire z3 spass e Isar proofs Try methods 100%

The output of the provers is as follows:

```
"cvc4": Try this: by (metis list.inject) (14 ms).
"z3": Try this: by (metis list.inject) (18 ms).
"spass": Try this: by (metis list.inject) (18 ms).
"e": Try this: by (metis the_elem_set) (14 ms).
"remote_vampire": Try this: by (metis list.inject) (16 ms).
```

The status bar at the bottom indicates the prover used is Sledgehammer, and the session information is (isabelle,isabelle,UTF-8-Isabelle)N m r o U C 188/31MB 12:14 AM.

Automated disprovers — counter examples

- **nitpick** based on **relational model finder**
- **quickcheck** based on **random functional evaluation**

The screenshot shows the Isabelle/Scratch IDE with a theory named 'Scratch'. The code defines a datatype 'tree' and two functions: 'tree_of_list' and 'list_of_tree'. A lemma 'tree_list (list_of_tree t) = t' is highlighted in yellow, and a counterexample is shown in a pop-up window.

```

theory Scratch
imports Main
begin

datatype 'a tree = Tip | Tree 'a "'a tree" "'a tree"

fun tree_of_list :: "'a list => 'a tree"
where
  "tree_of_list [] = Tip"
| "tree_of_list (x # xs) = Tree x Tip (tree_of_list xs)"

fun list_of_tree :: "'a tree => 'a list"
where
  "list_of_tree Tip = []"
| "list_of_tree (Tree x t1 t2) = x # list_of_tree t1 @ list_of_tree t2"

lemma "list_of_tree (tree_of_list xs) = xs"
  by (induct xs) simp_all

lemma "tree_list (list_of_tree t) = t"

```

Auto Quickcheck found a counterexample:
 t = Tree a₁ (Tree a₁ Tip Tip) Tip
 Evaluated terms:
 tree_list (list_of_tree t) =
 Tree a₁ Tip (Tree a₁ Tip Tip)

Query
 20,42 (476/477) Input/output complete (Isabelle, Isabelle, UTF-8-Isabelle) 6/391MB 11:38 PM

Isabelle/HOL proof methods

- *rule*: generic Natural Deduction (with HO unification)
- *cases*: elimination, syntactic representation of datatypes, inversion of inductive sets and predicates
- *induct* and *coinduct*: induction and coinduction of types, sets, predicates
- *simp*: equational reasoning by the Simplifier (HO rewriting), with possibilities for add-on tools
- *fast* and *blast*: classical reasoning (tableau)
- *auto* and *force*: combined simplification and classical reasoning
- *arith*, *presburger*: specific theories
- *smt*: Z3 with proof reconstruction

Theory specifications

Types

- augmented version of **Simple Theory of Types** (Church 1940)
- schematic **polymorphism** (weaker than ML let-polymorphism)
- basic types: *bool*, *nat*, $'a \Rightarrow 'b$ (full function space)

Type specifications:

- **typedef** semantic subtype of existing type
- **quotient_type** wrt. equivalence relation or PER
- **record** extensible records (glorified tuples)
- **datatype** and **codatatype** (**Bounded Natural Functors**)

datatype $'a$ *list* = *Nil* | *Cons* (*hd*: $'a$) (*tl*: $'a$ *list*)

codatatype $'a$ *stream* = *Stream* (*HD*: $'a$) (*TL*: $'a$ *stream*)

Functions and constants

- **abbreviation:** **abstract syntax** definitions

abbreviation (*input*) $double :: nat \Rightarrow nat$
where $double\ n \equiv 2 * n$

- **definition:** simple **non-recursive** definitions

definition $square :: nat \Rightarrow nat$
where $square\ n = n * n$

- **fun** and **function** / **termination:** **general recursion** with implicit or explicit termination proof

fun $fibonacci :: nat \Rightarrow nat$
where
 $fibonacci\ 0 = 0$
 $| fibonacci\ (Suc\ 0) = 1$
 $| fibonacci\ (Suc\ (Suc\ n)) = fibonacci\ n + fibonacci\ (Suc\ n)$

Inductive predicates and sets

- **inductive** and **coinductive**: Knaster-Tarski fixed-points over predicates or sets

inductive_set *star* ($_ \star$ [100] 100) **for** $R :: ('a \times 'a)$ set

where

base: $(x, x) \in R \star$

| *step*: $(x, y) \in R \implies (y, z) \in R \star \implies (x, z) \in R \star$

This means $R \star$ is the least relation (set of pairs) that is closed under the **introduction rules** above. The following **induction rule** is provided:

$(x_1, x_2) \in R \star \implies$

$(\bigwedge x. P x x) \implies$

$(\bigwedge x y z. (x, y) \in R \implies (y, z) \in R \star \implies P y z \implies P x z) \implies P x_1 x_2$

Type classes

- Predicate over **constant signature** with **single type-variable**
- Integrated into type-system: **order-sorted algebra** of constraints
- Class intersections are called **sorts**
- Class inclusion **hierarchy**: by definition or proof
- Class instantiation by **concrete types**

```
class zero = fixes zero :: 'a (0)
```

```
class one = fixes one :: 'a (1)
```

```
class times = fixes times :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a (infixl * 70)
```

```
class group = times + one + inverse +
```

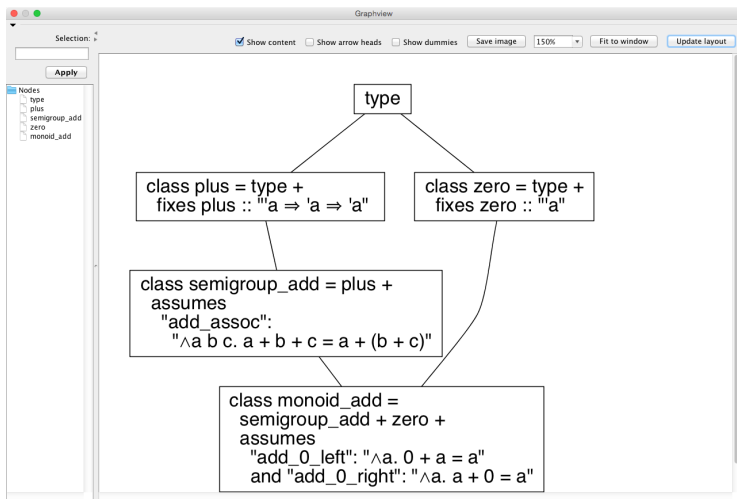
```
  assumes group_assoc: (x * y) * z = x * (y * z)
```

```
  and group_left_one: 1 * x = x
```

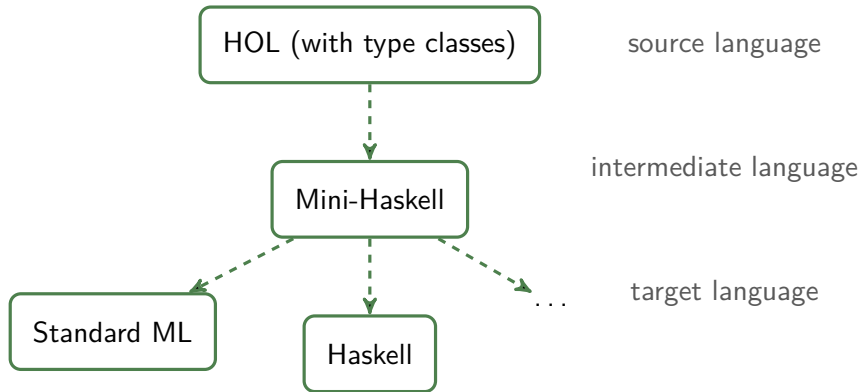
```
  and group_left_inverse: inverse x * x = 1
```

Example: class hierarchy

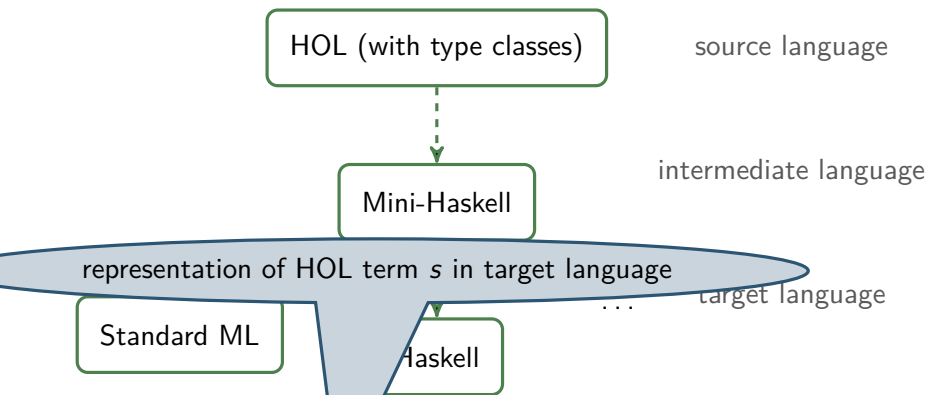
`class_deps` *type monoid_add*



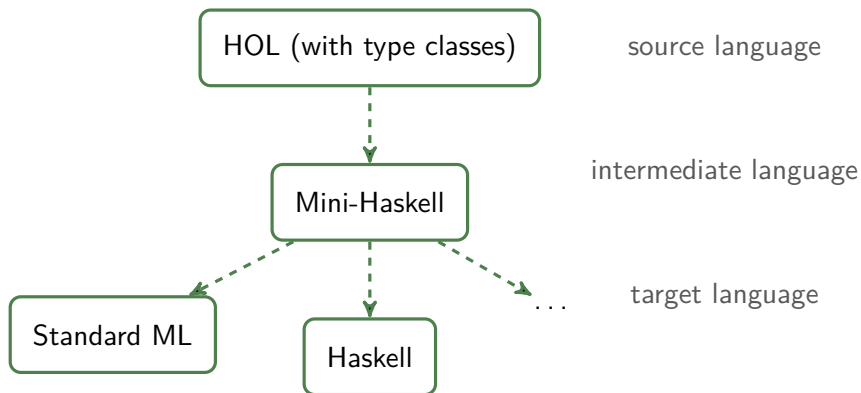
Code generation



- sound, provided target language has **rewriting semantics** (i.e. evaluation $[s] \rightsquigarrow [t]$ implies $s = t$ in HOL)



- sound, provided target language has **rewriting semantics** (i.e. evaluation $[s] \rightsquigarrow [t]$ implies $s = t$ in HOL)



- sound, provided target language has **rewriting semantics** (i.e. evaluation $[s] \rightsquigarrow [t]$ implies $s = t$ in HOL)
- only partial correctness (generated functions might diverge)

Code equations

- specify how to “interpret” constants in generated code
- registered and modified via attribute *code*
- provided automatically by many commands:
definition, **fun**, **function** etc.

Triggering code generation

- specify **where** and in what **target language** code should be generated for given **constants**
- **export_code**

Code equations

- specify how to “interpret” constants in generated code
- registered and modified via attribute *code*
- provided automatically by many commands:
definition, **fun**, **function** etc.

Triggering code generation

- specify **where** and in what **target language** code should be generated for given **constants**
- **export_code**

Example

```
export_code rev in SML (*file "/tmp/rev.ML"*)
```

Structured proofs: Isabelle/Isar

Atomic proofs

- Skipped proofs:

sorry

- Single-step proofs:

by *fact*

by *this* \equiv .

by *rule* \equiv ..

- Automated proofs:

by *auto*

by *simp*

by *blast*

by *force*

Goal refinement

- Canonical double-step:

```
have prop by (initial_method) (terminal_method)
```

or:

```
have prop  
proof (initial_method)  
qed (terminal_method)
```

- Structured proof body:

```
have prop  
proof (initial_method)  
  fix vars  
  assume props  
  show prop ⟨proof⟩  
qed (terminal_method)
```

```
have prop  
proof (initial_method)  
  case (case_name vars)  
  show prop ⟨proof⟩  
qed (terminal_method)
```

Using facts

- Goal statement with facts:

```
from facts1 have prop using facts2  
proof (initial_method)  
  body  
qed (terminal_method)
```

- *initial_method* sees *facts*₁ *facts*₂ as primary argument
- actual use of facts depends on proof method,
e.g. *rule*, *cases*, *induct*, *auto*

- Abbreviations and synonyms:

```
from this   ≡ then  
from a     ≡ note a then  
with a    ≡ note a and this then
```

Example

notepad

begin

fix $A B :: \text{bool}$

have $A \wedge B \longrightarrow B \wedge A$

proof (*rule impI*)

assume $*$: $A \wedge B$

show $B \wedge A$

proof (*rule conjI*)

from $*$ **show** B **by** (*rule conjunct2*)

from $*$ **show** A **by** (*rule conjunct1*)

qed

qed

end

Context elements

- Universal context: **fix** and **assume**

$$\left\{ \begin{array}{l} \mathbf{fix} \ x \\ \mathbf{have} \ B \ x \ \mathbf{sorry} \\ \end{array} \right\}$$

have $\bigwedge x. B \ x$ **by fact**

$$\left\{ \begin{array}{l} \mathbf{assume} \ A \\ \mathbf{have} \ B \ \mathbf{sorry} \\ \end{array} \right\}$$

have $A \implies B$ **by fact**

- Existential context: **obtain**

$$\left\{ \begin{array}{l} \mathbf{obtain} \ a \ \mathbf{where} \ B \ a \ \mathbf{sorry} \\ \mathbf{have} \ C \ \mathbf{sorry} \\ \end{array} \right\}$$

have C **by fact**

Structured statements

- prefix of **context elements**: **fixes**, **assumes**, ...
- regular **conclusion shows**
- **dual conclusion obtains** — “may assume that ...”

theorem \exists *intro*:

fixes $a :: 'a$ **and** $B :: 'a \Rightarrow bool$

assumes $B a$

shows $\exists x. B x$

theorem \exists *elim*:

fixes $B :: 'a \Rightarrow bool$

assumes $\exists x. B x$

obtains $a :: 'a$ **where** $B a$

- useful diagnostic command: **print_statement**