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

Tobias Nipkow:
Isabelle/HOL 1993

1969 — 1993

1972 **Robin Milner:**
Stanford LCF

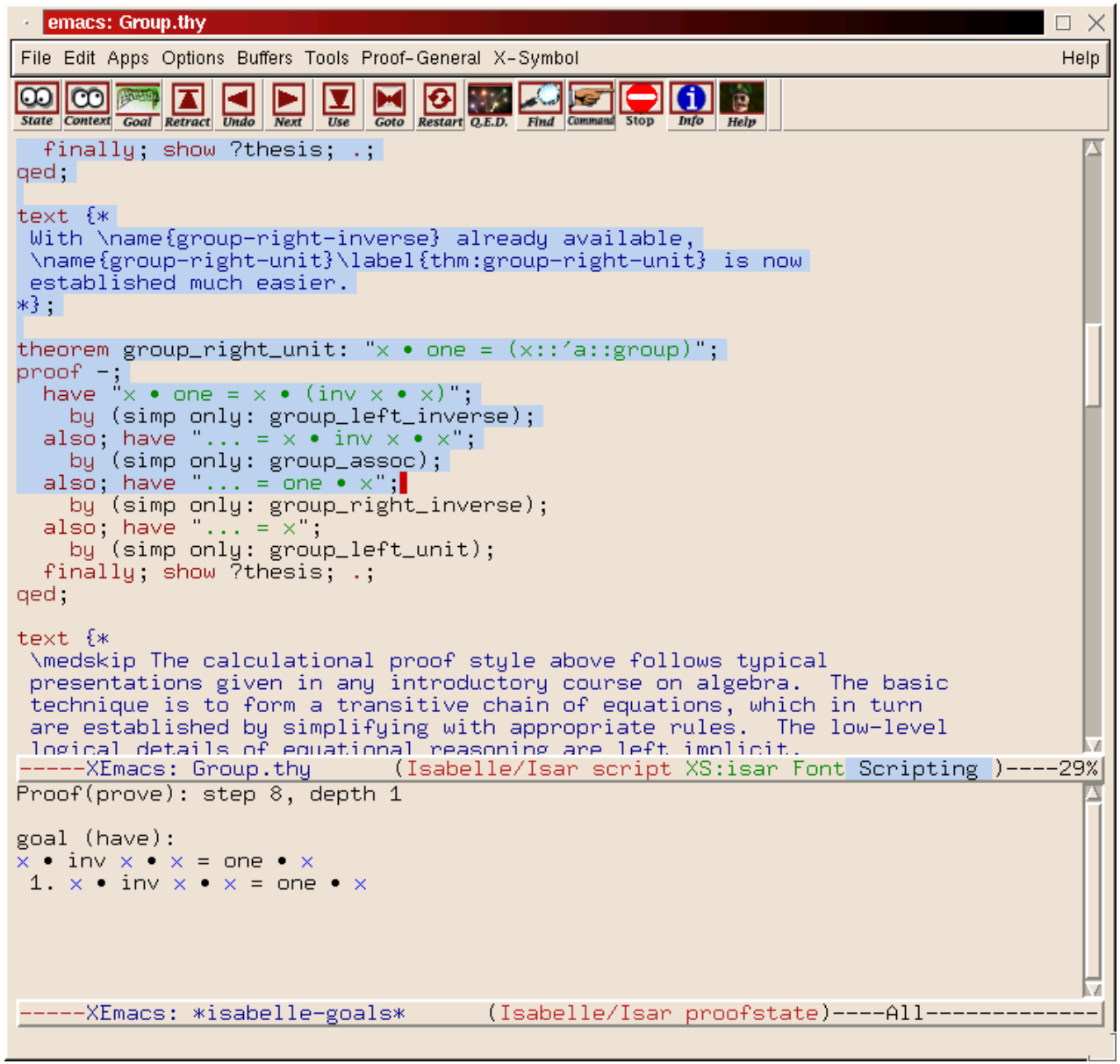
1977 **Edinburgh LCF**

1986 **Isabelle**

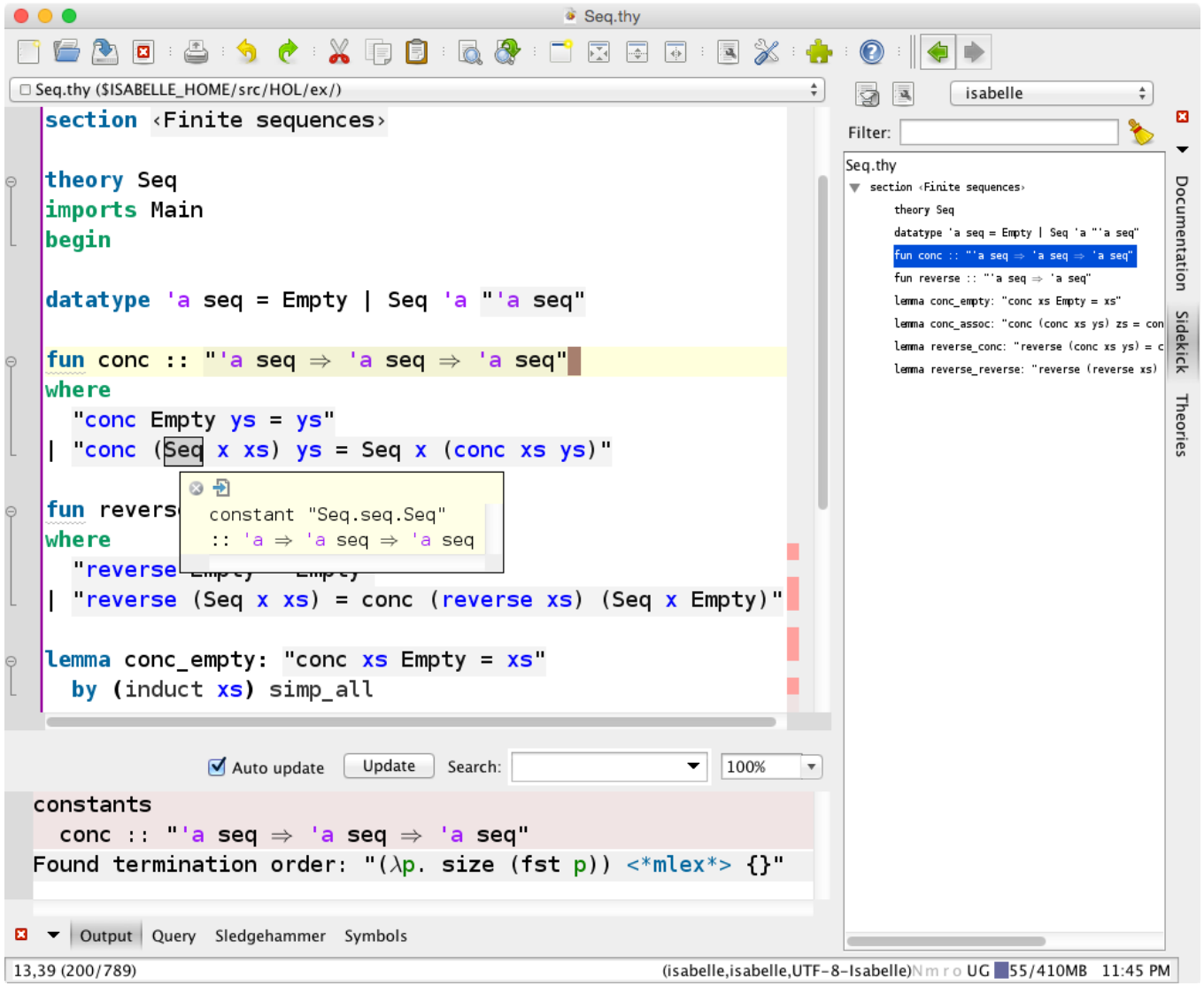
TTY interaction (1979)

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

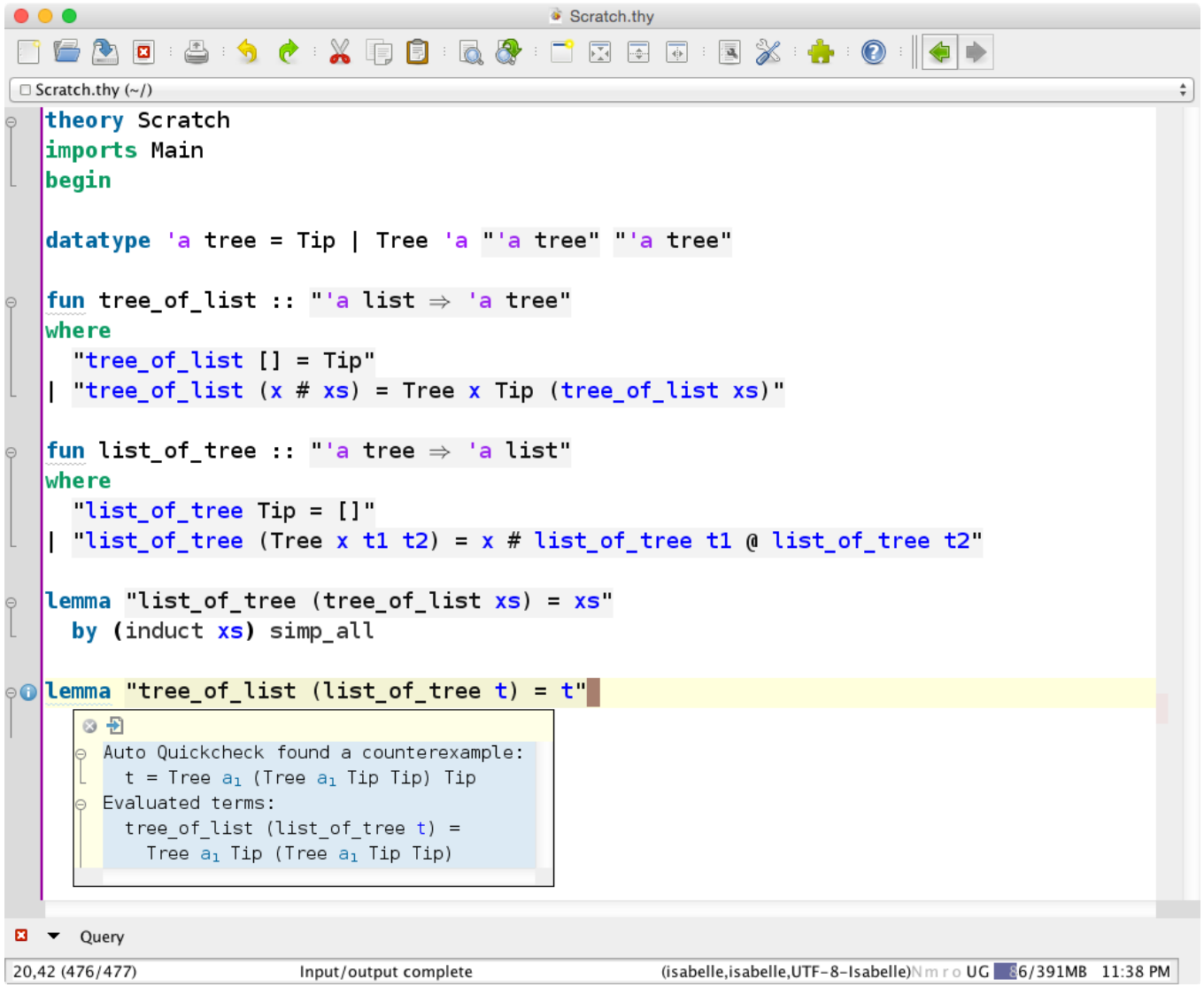
Proof General / Emacs interaction (1999)



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

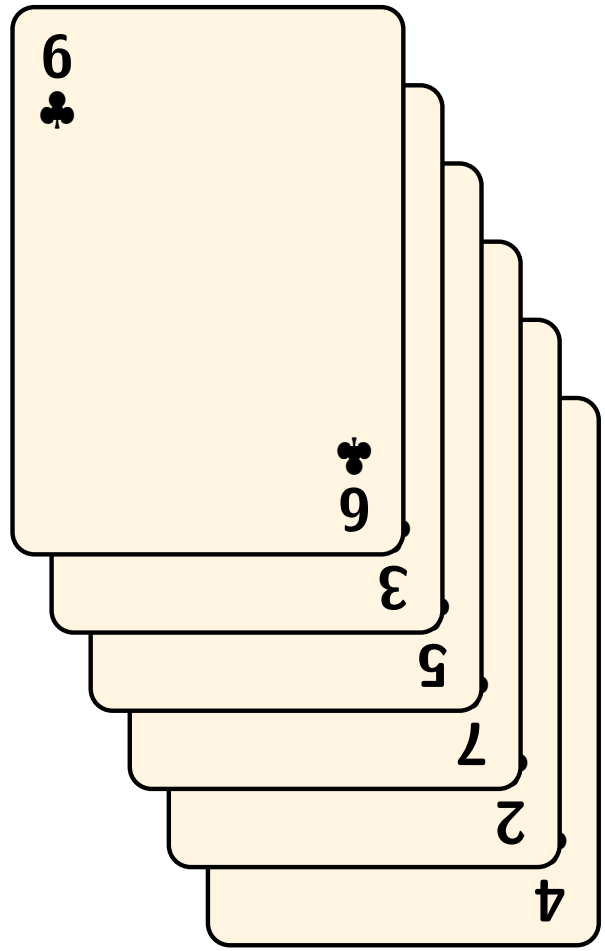


Orchestration of (dis)provers in Isabelle/jEdit

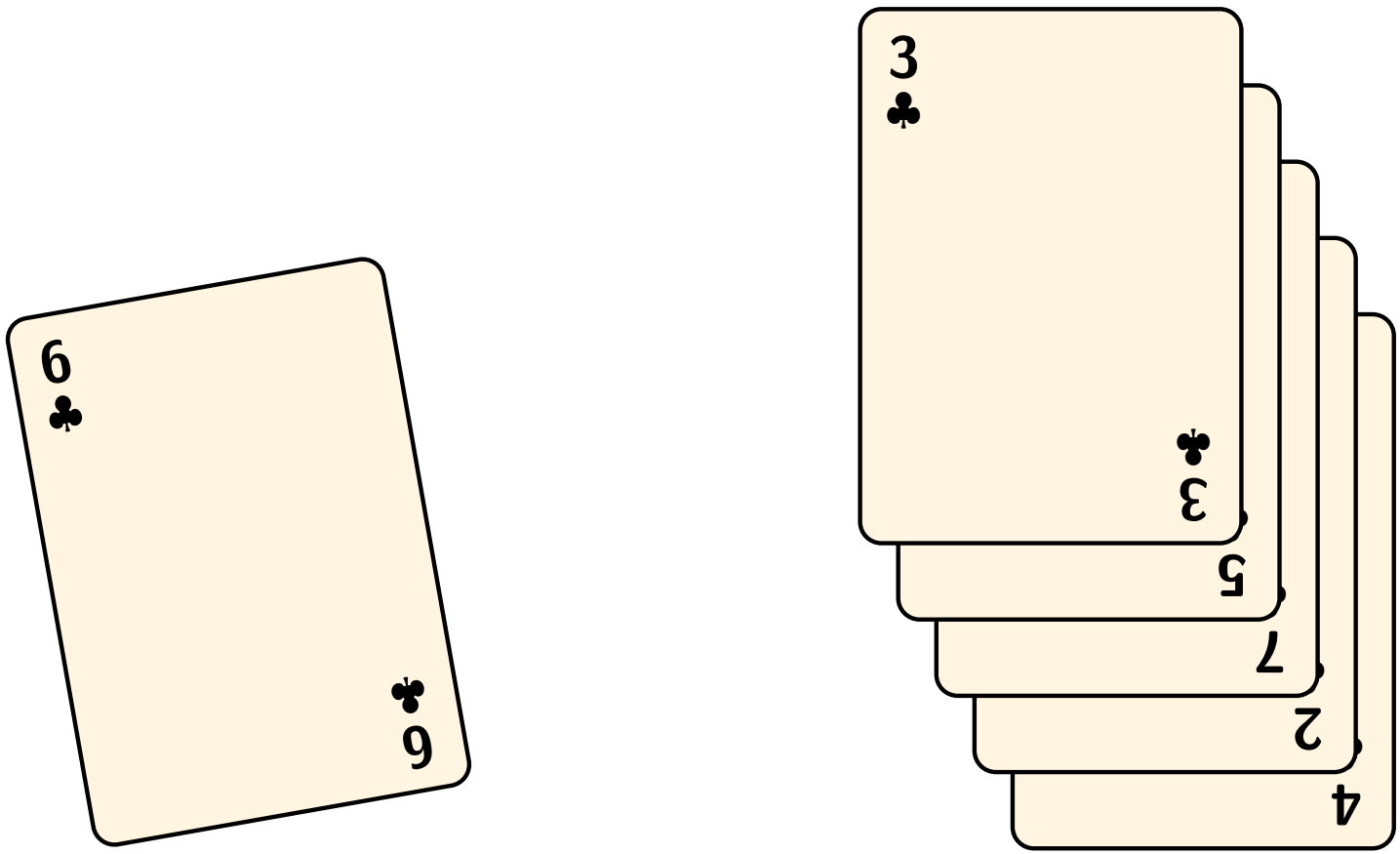


Functional programming in Isabelle/HOL

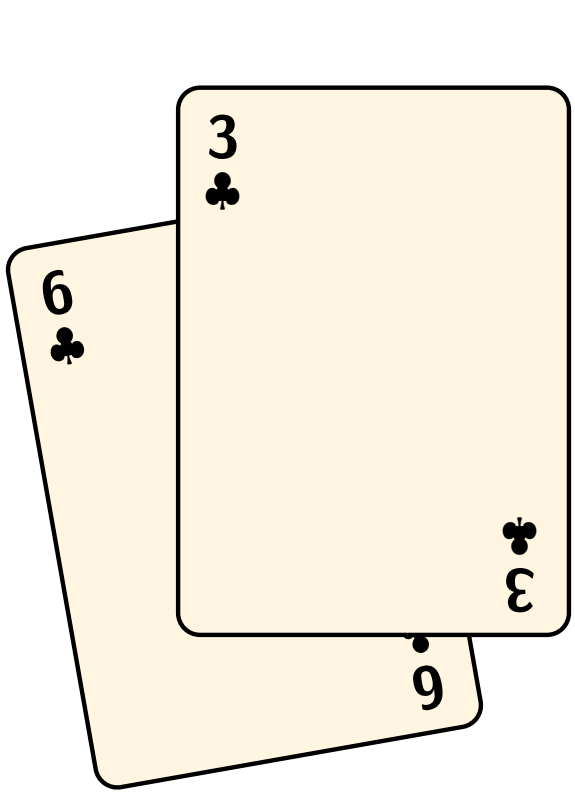
Example: Insertion Sort

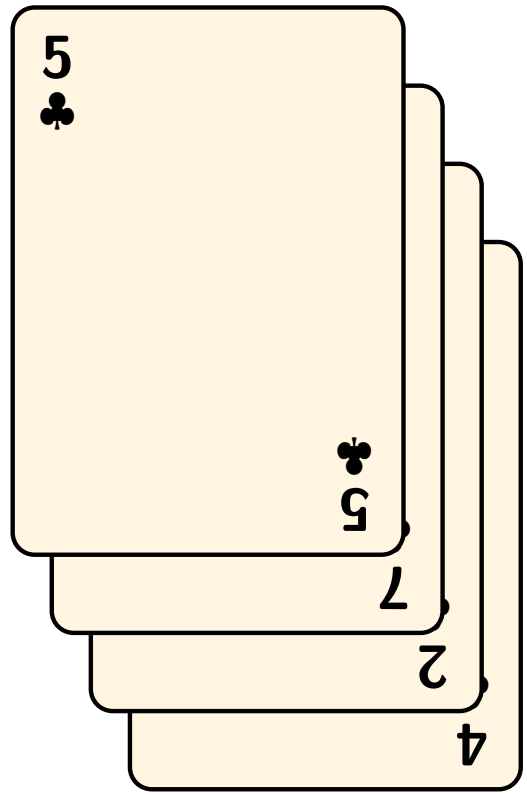


Example: Insertion Sort

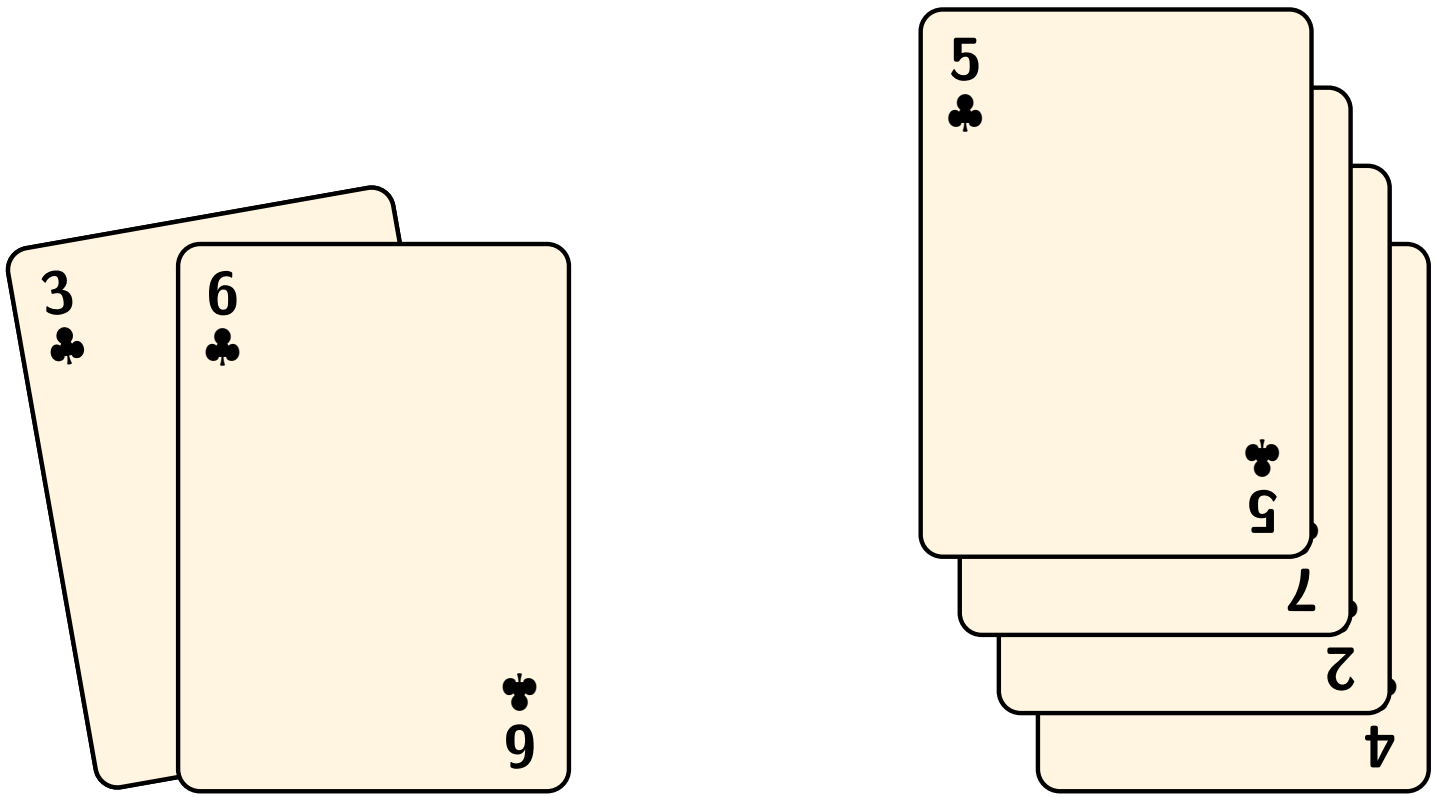


Example: Insertion Sort

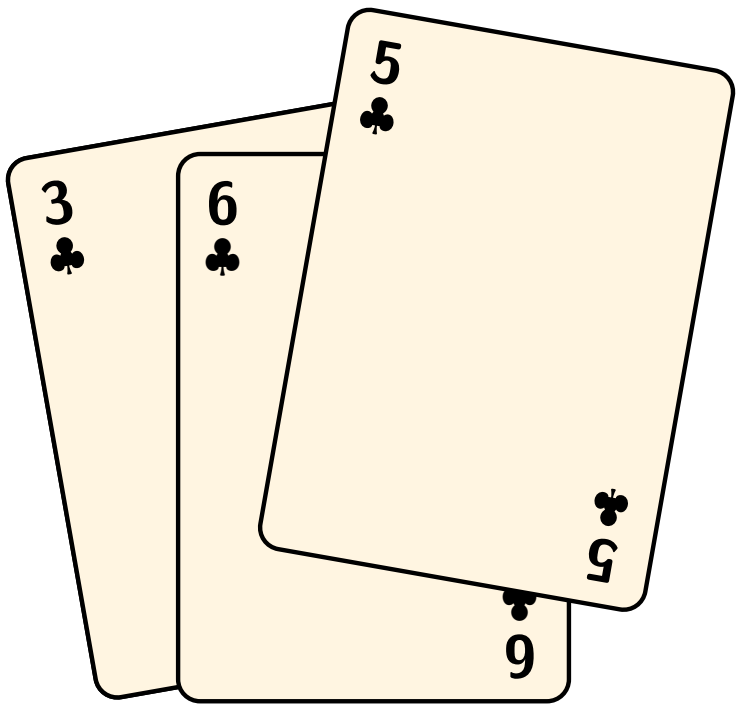


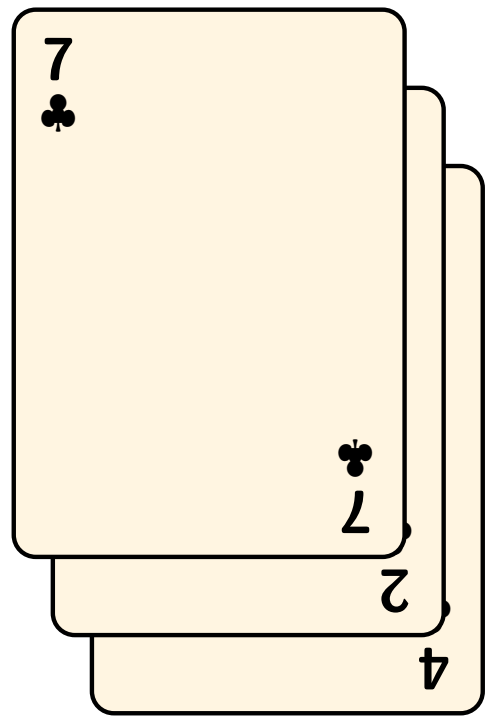


Example: Insertion Sort

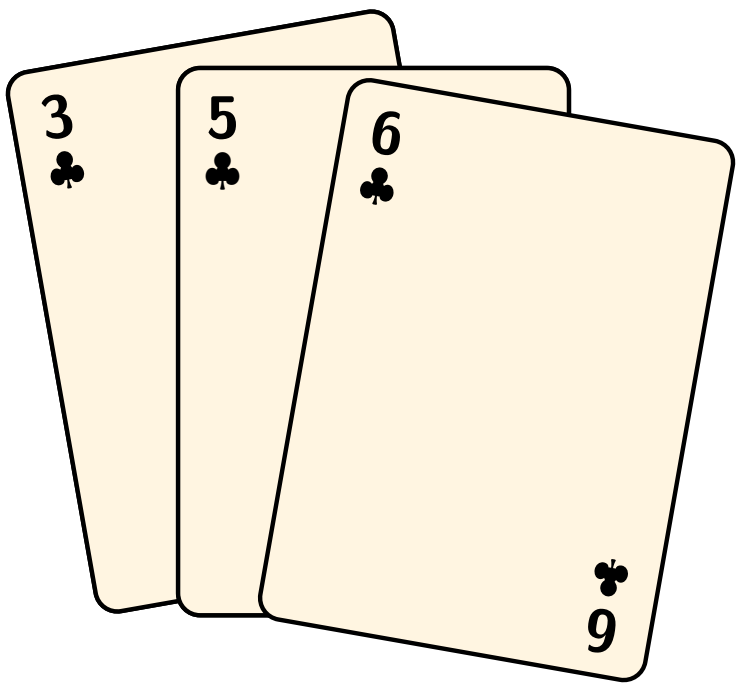


Example: Insertion Sort

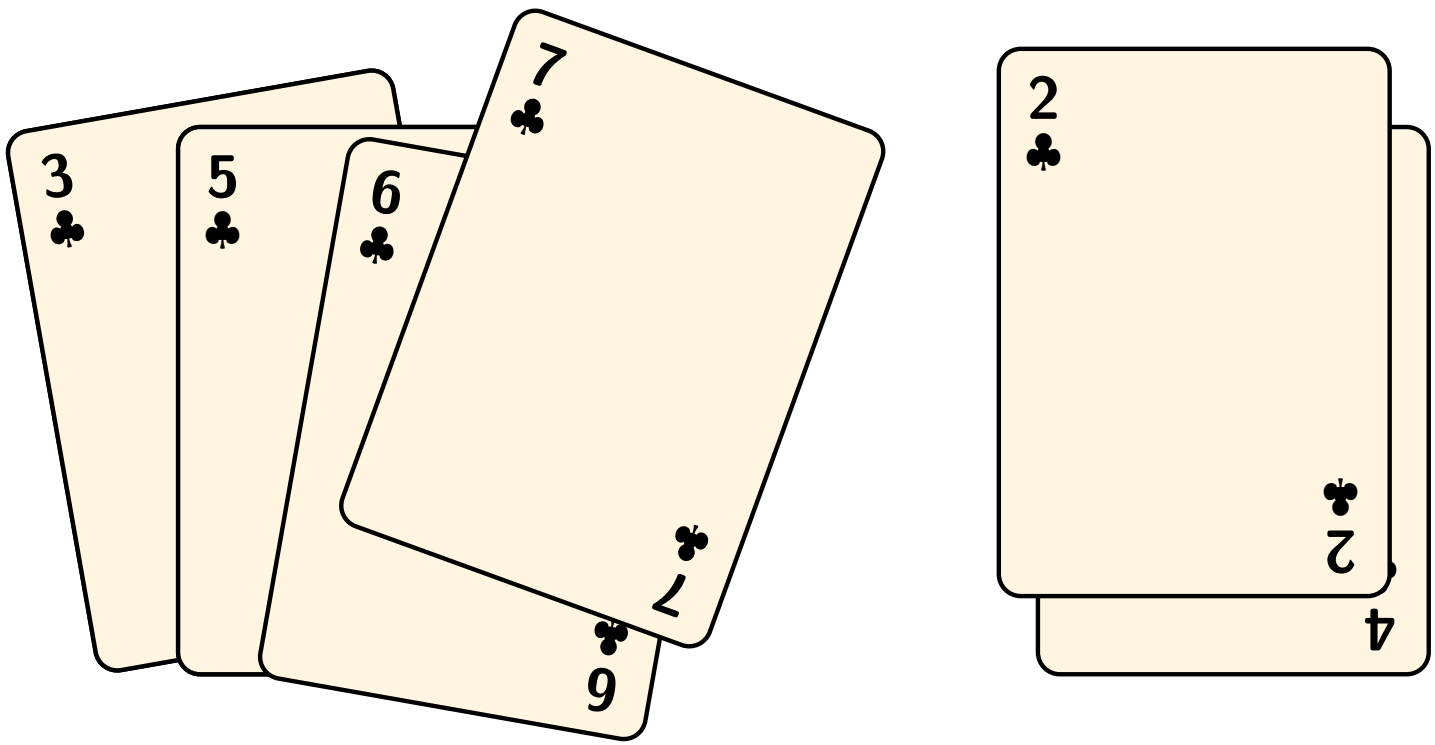




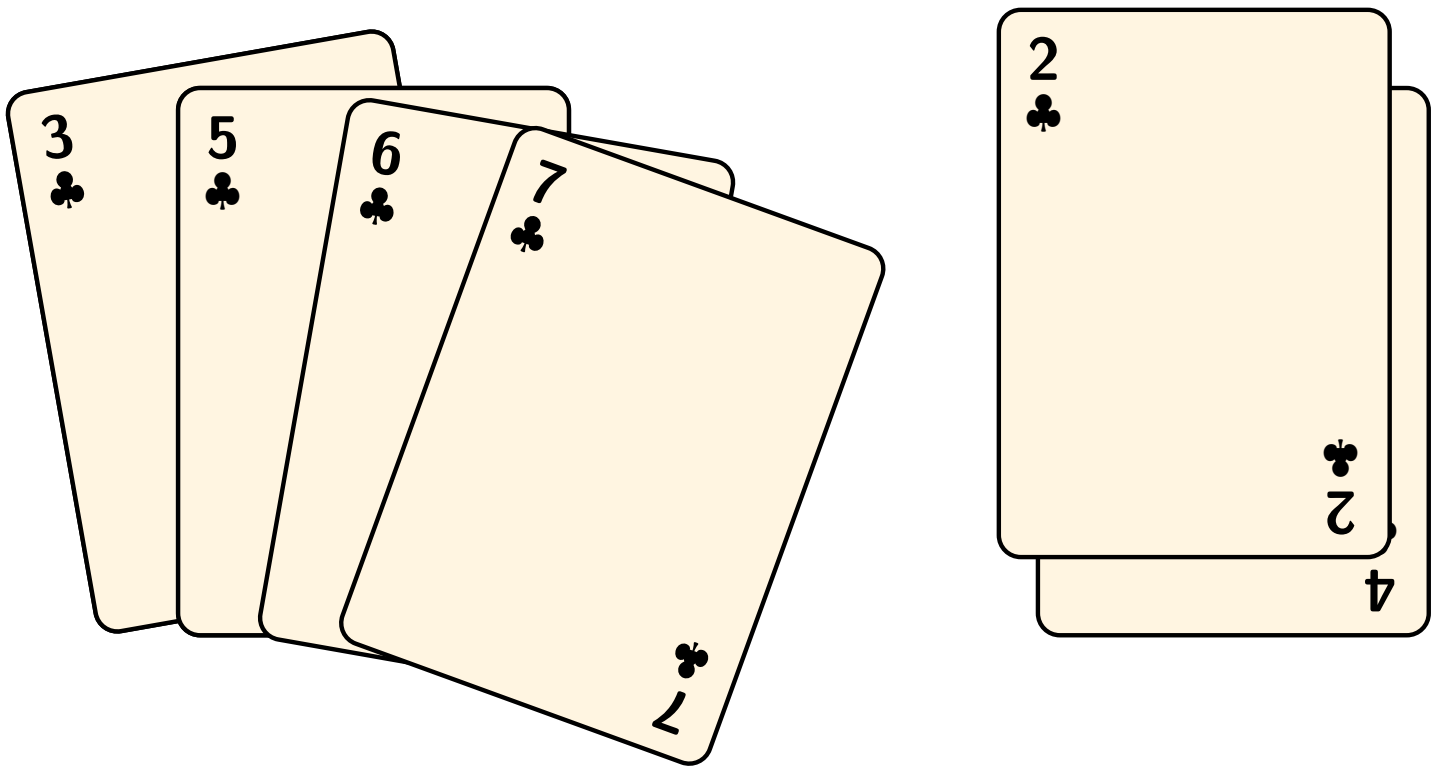
Example: Insertion Sort



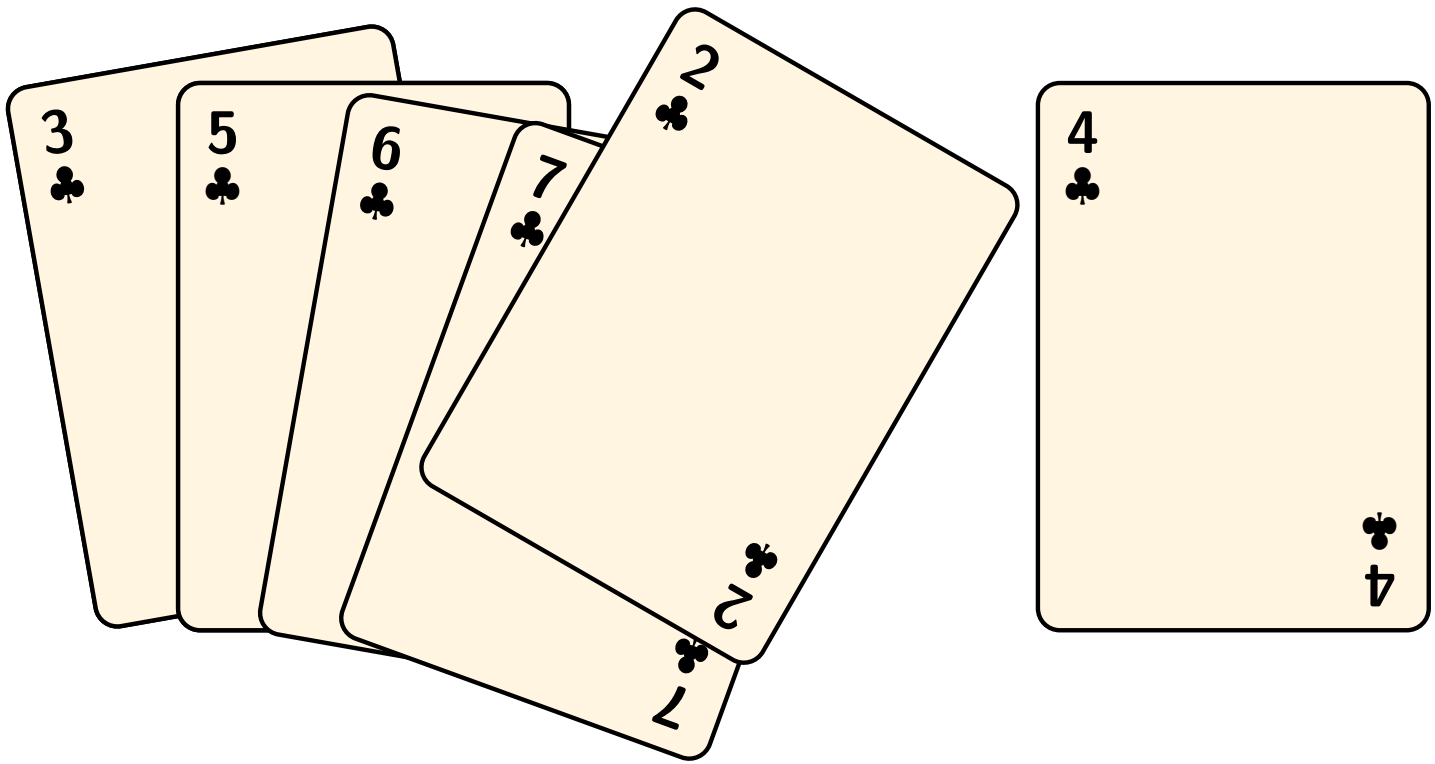
Example: Insertion Sort



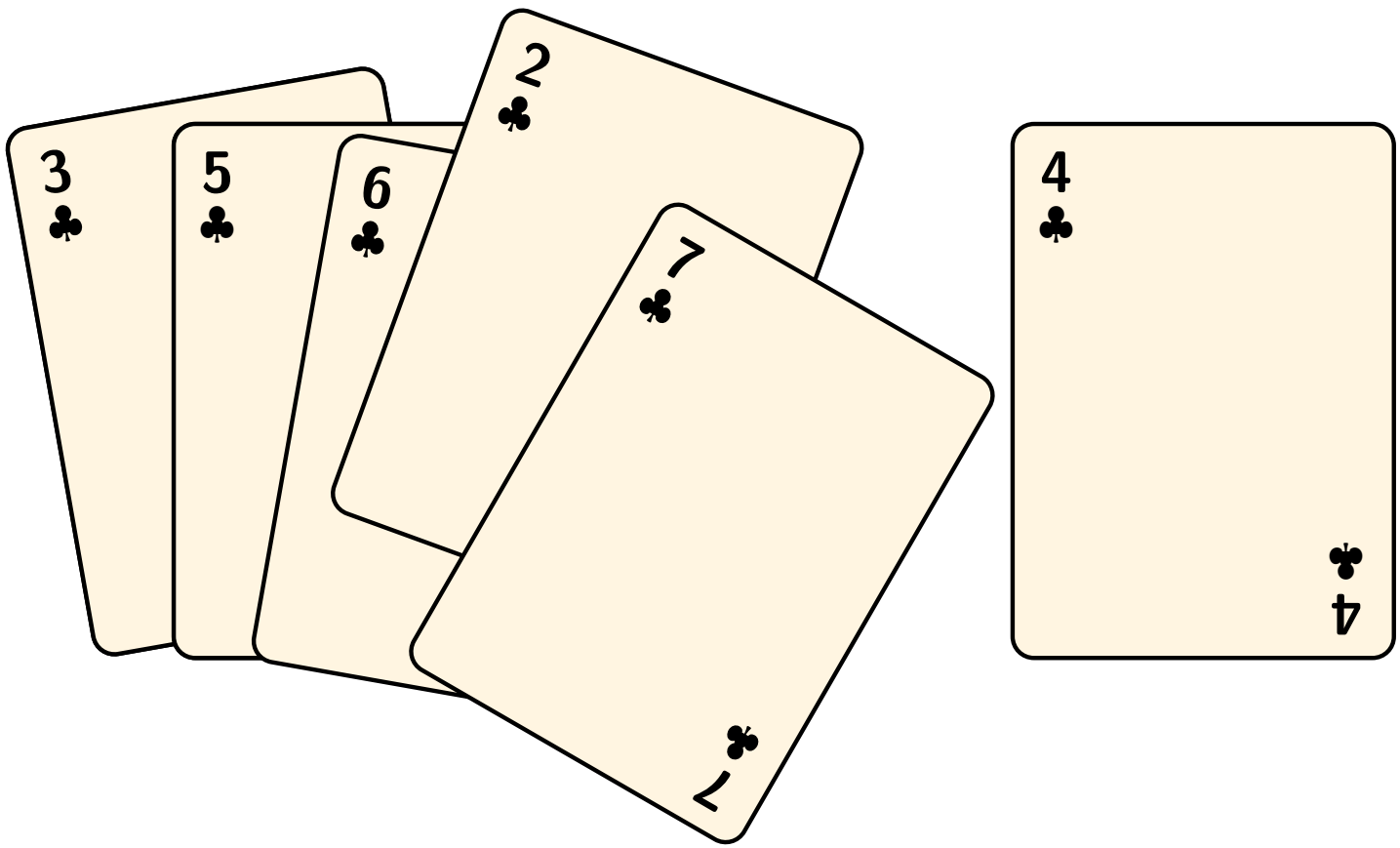
Example: Insertion Sort



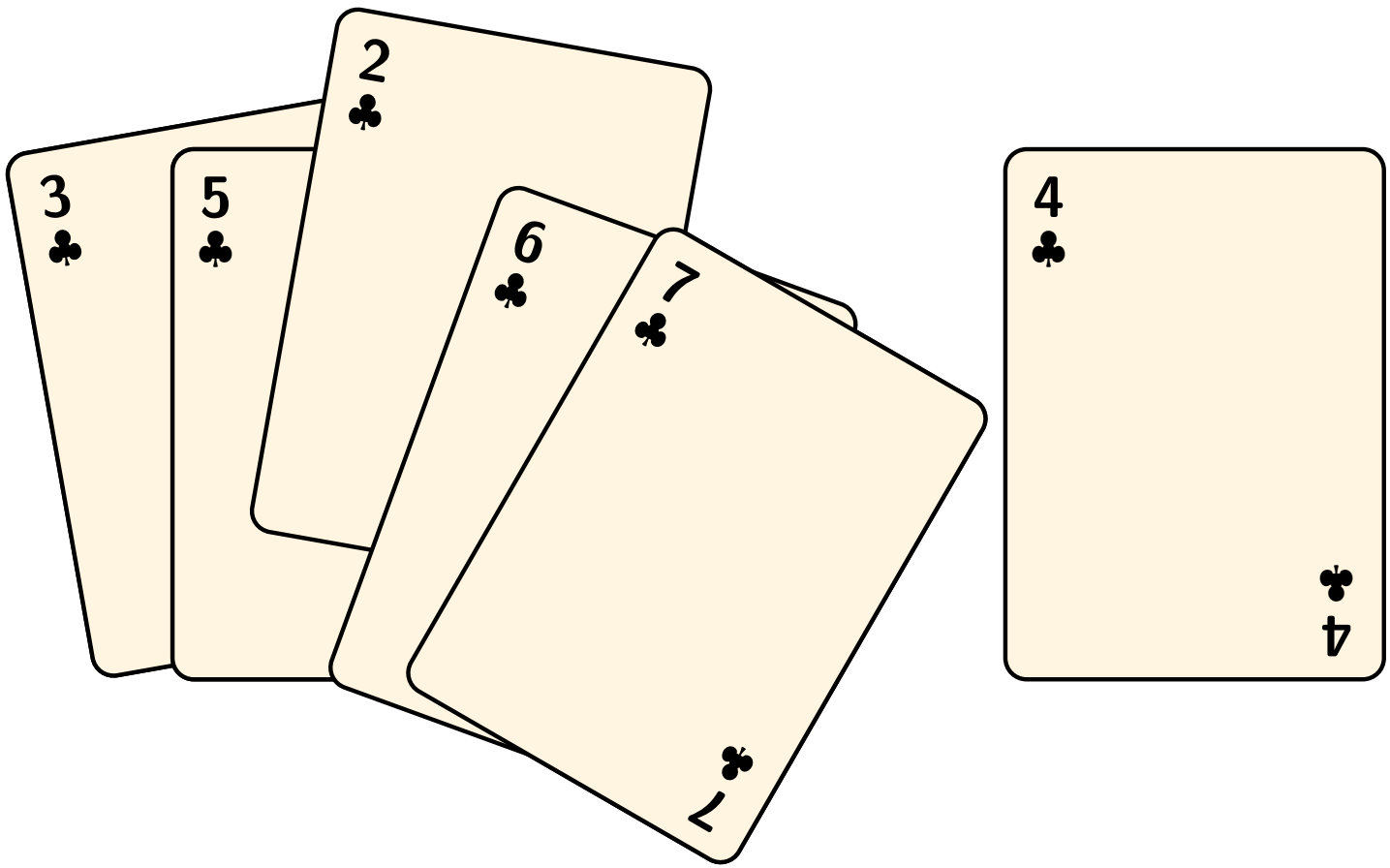
Example: Insertion Sort



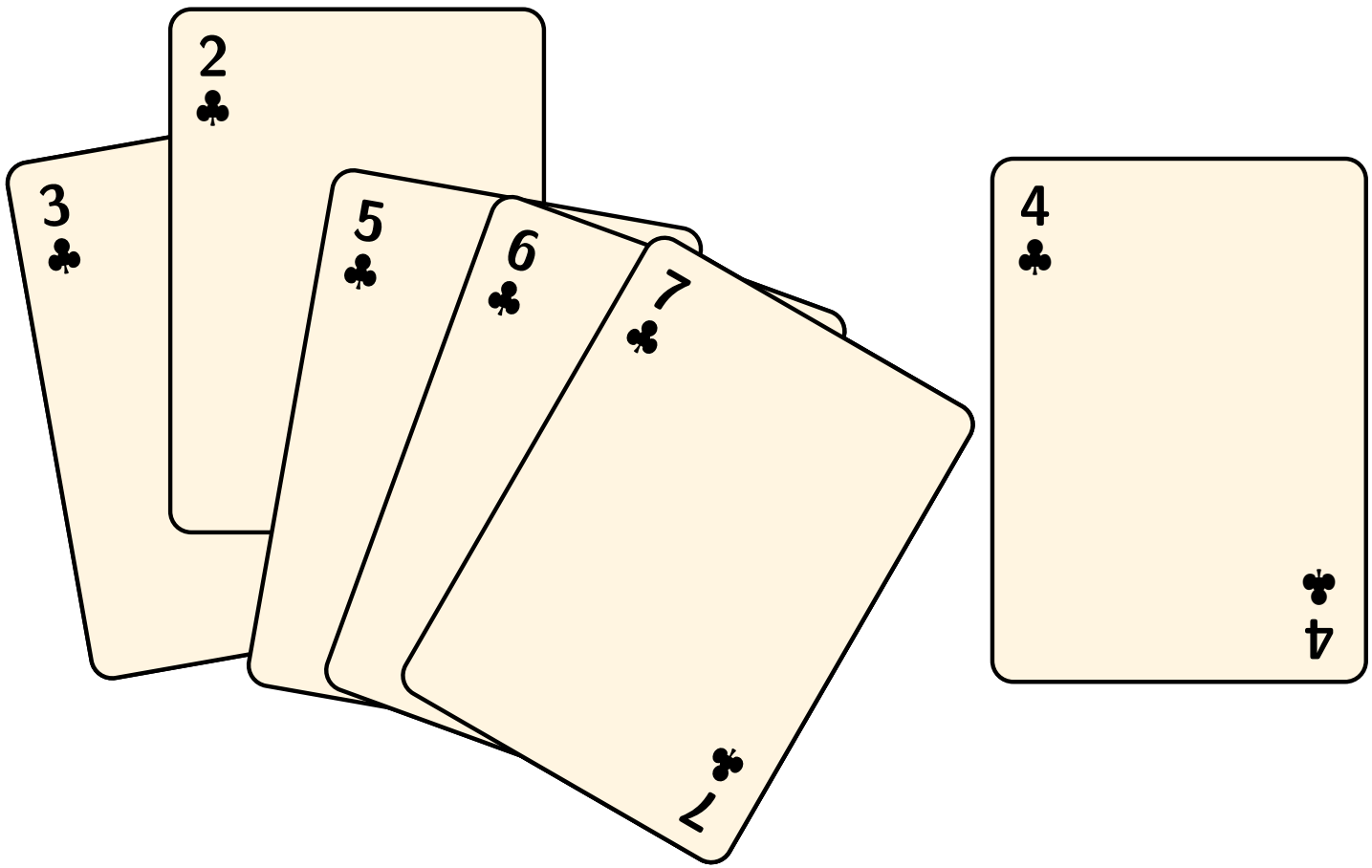
Example: Insertion Sort



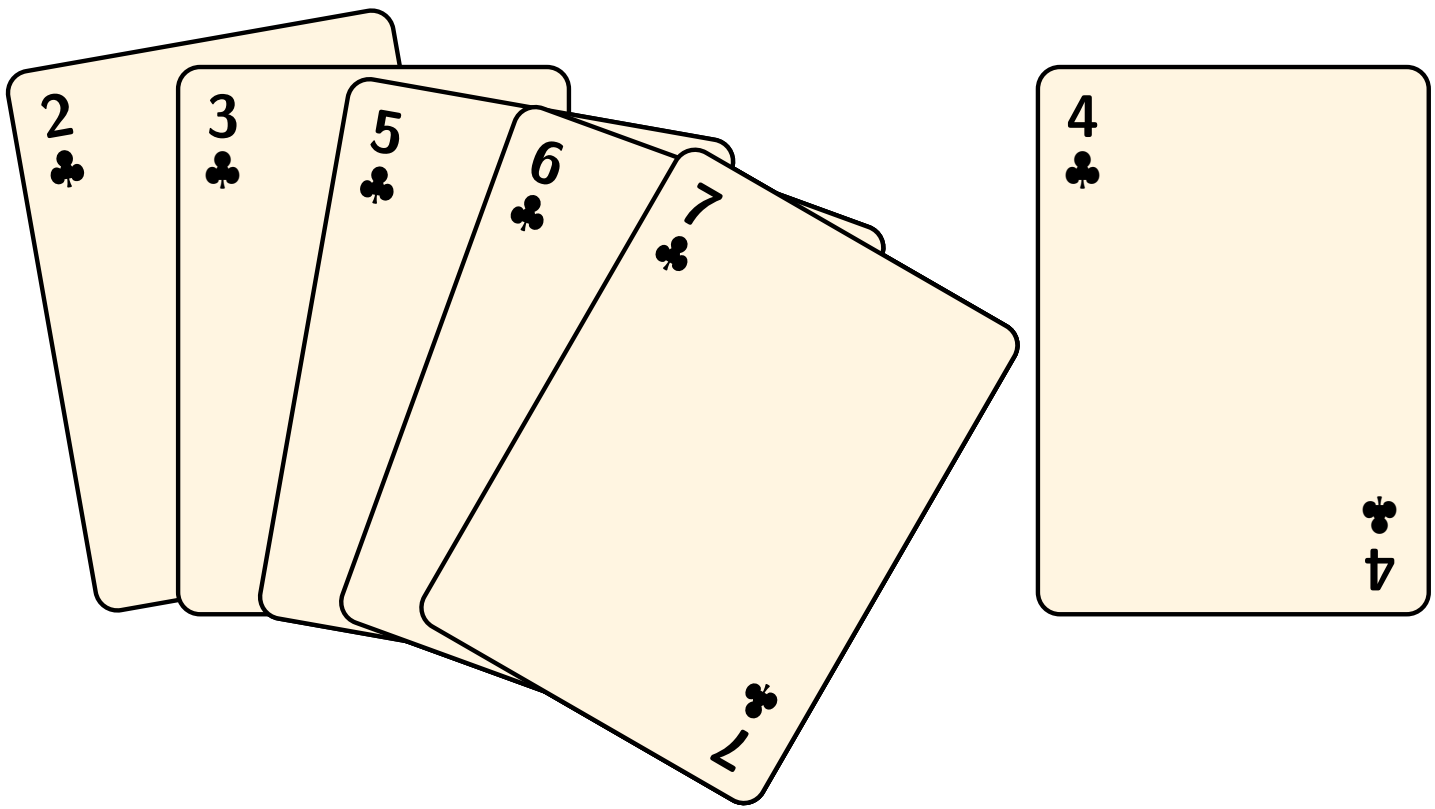
Example: Insertion Sort



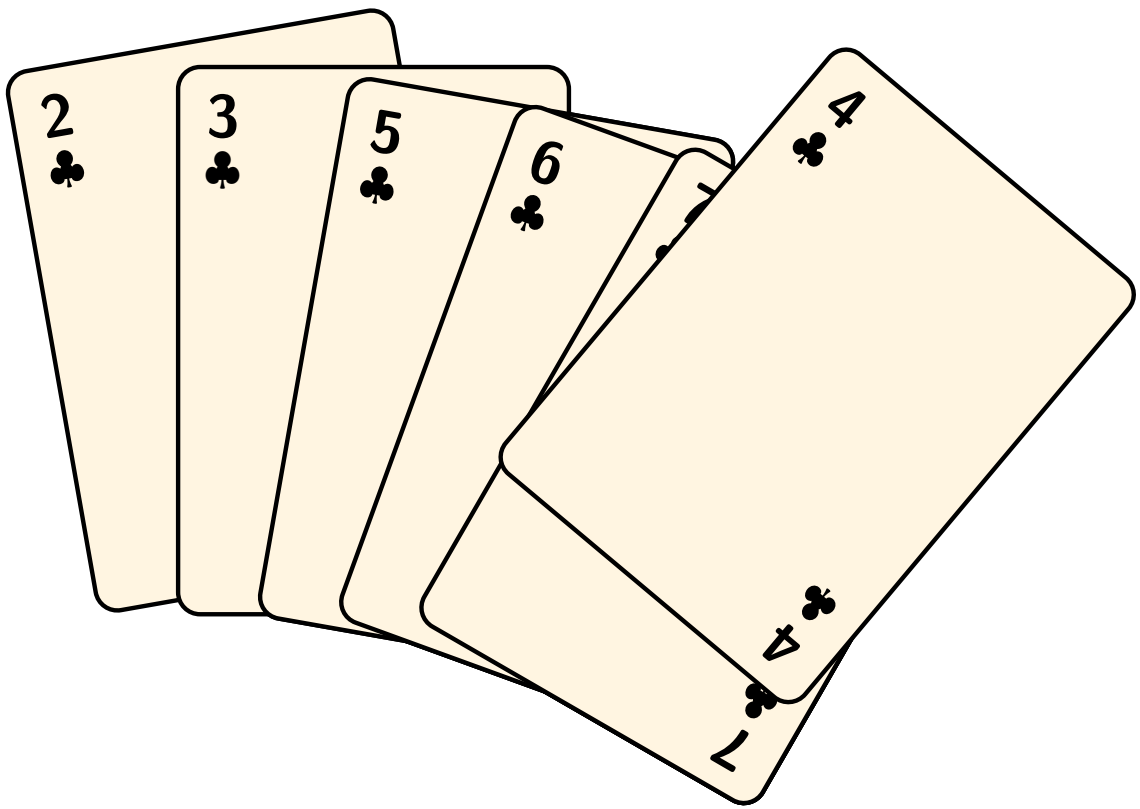
Example: Insertion Sort



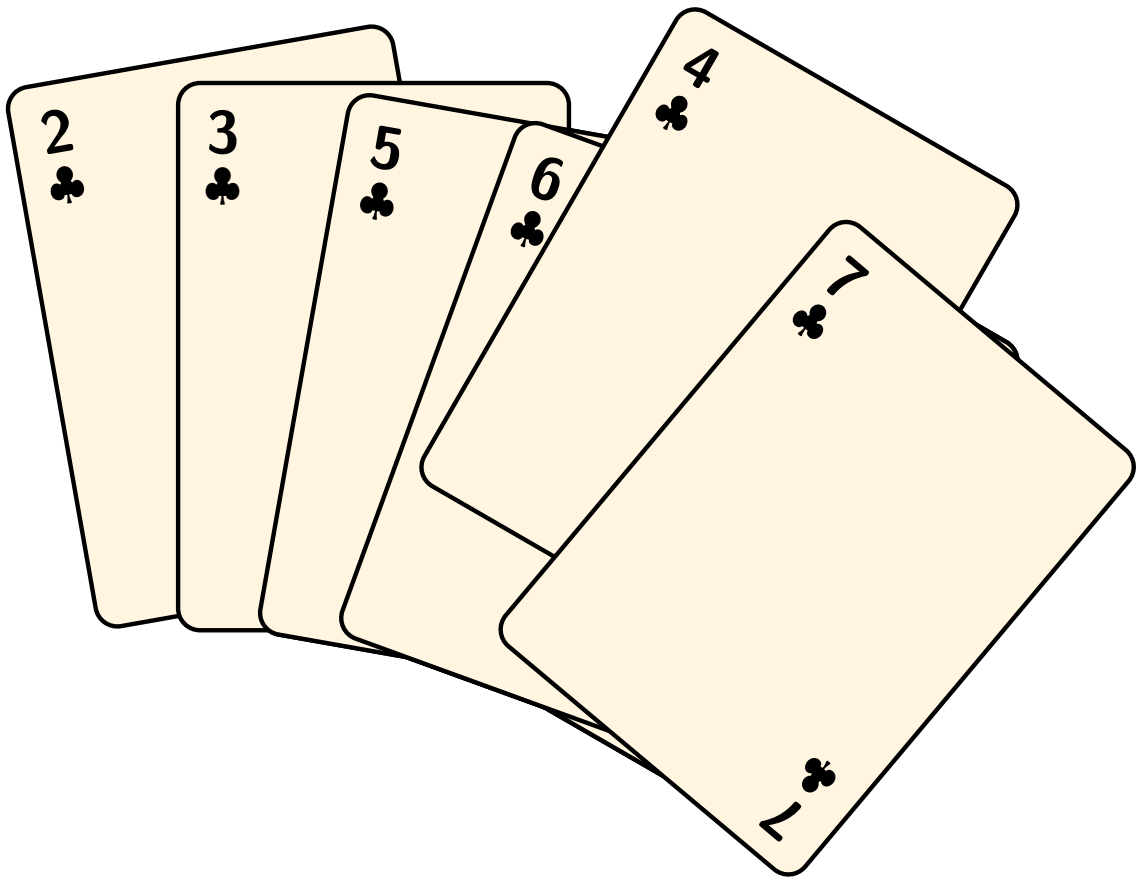
Example: Insertion Sort



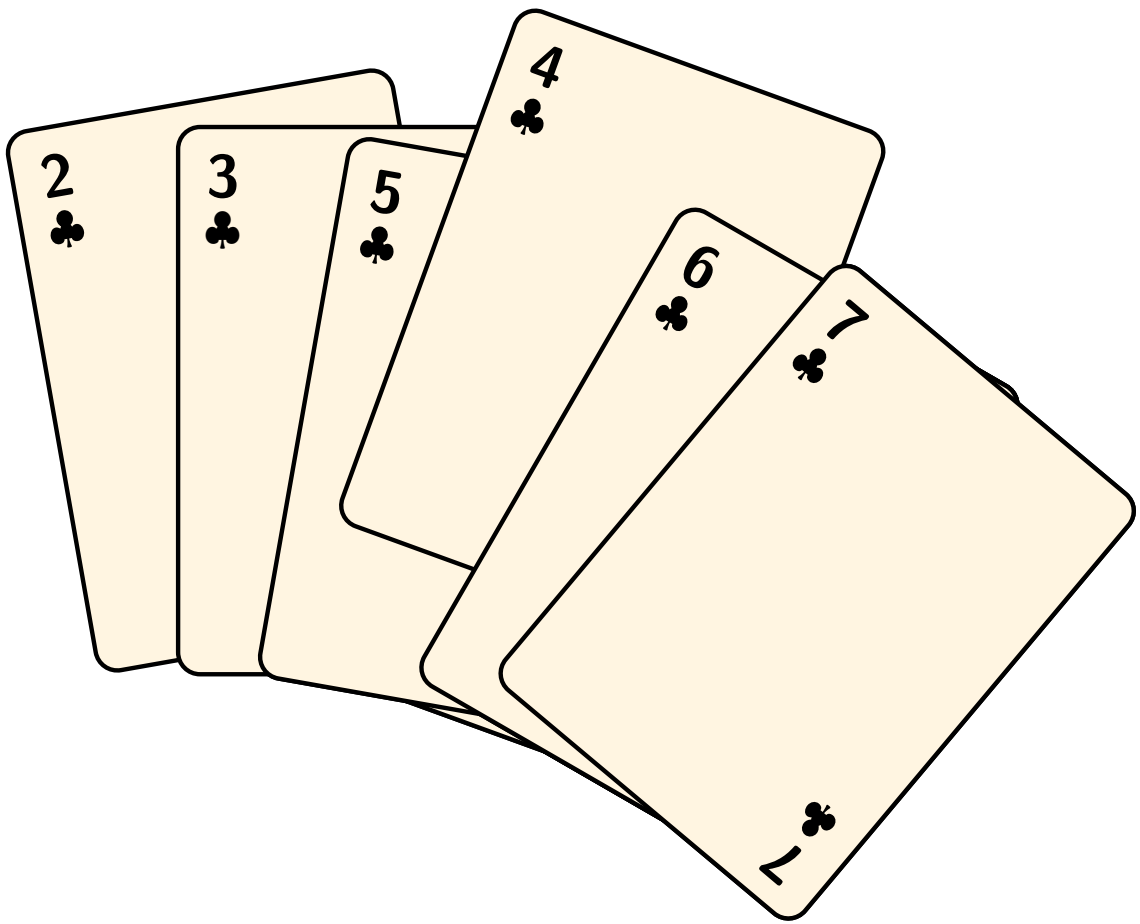
Example: Insertion Sort



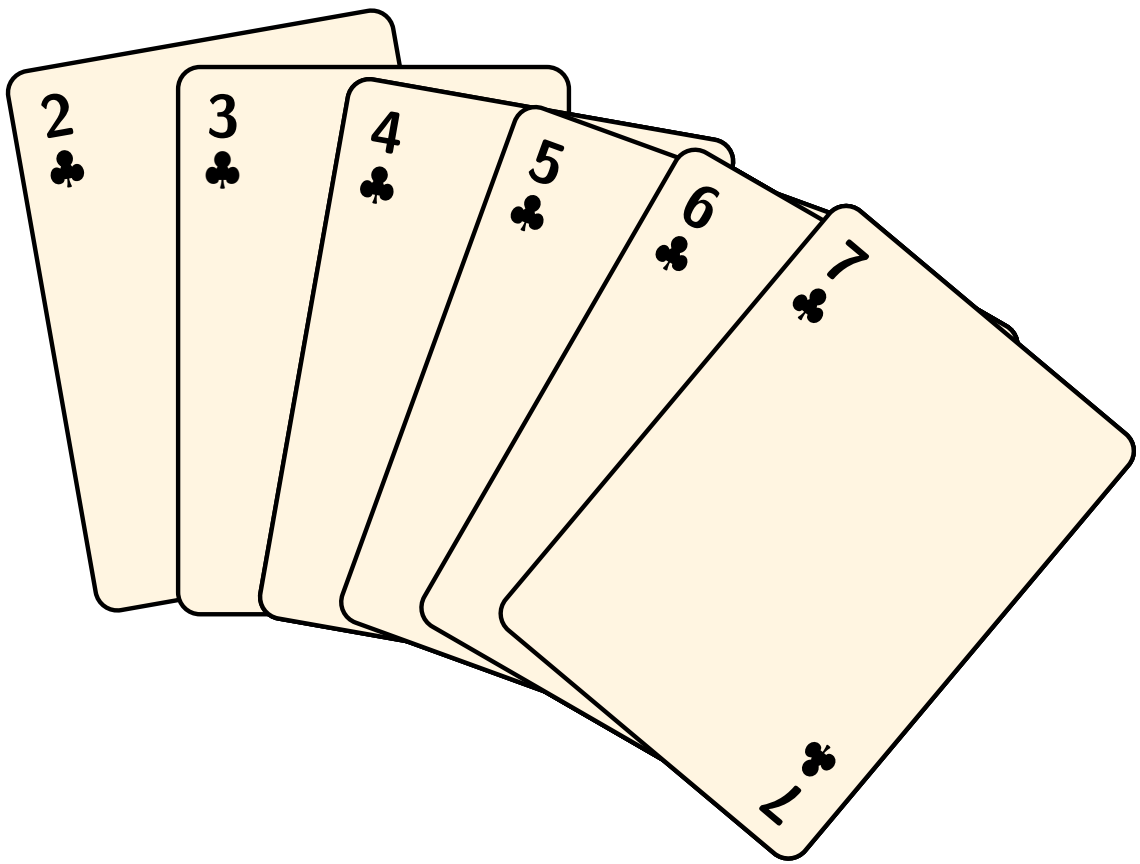
Example: Insertion Sort



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

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

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

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

- 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

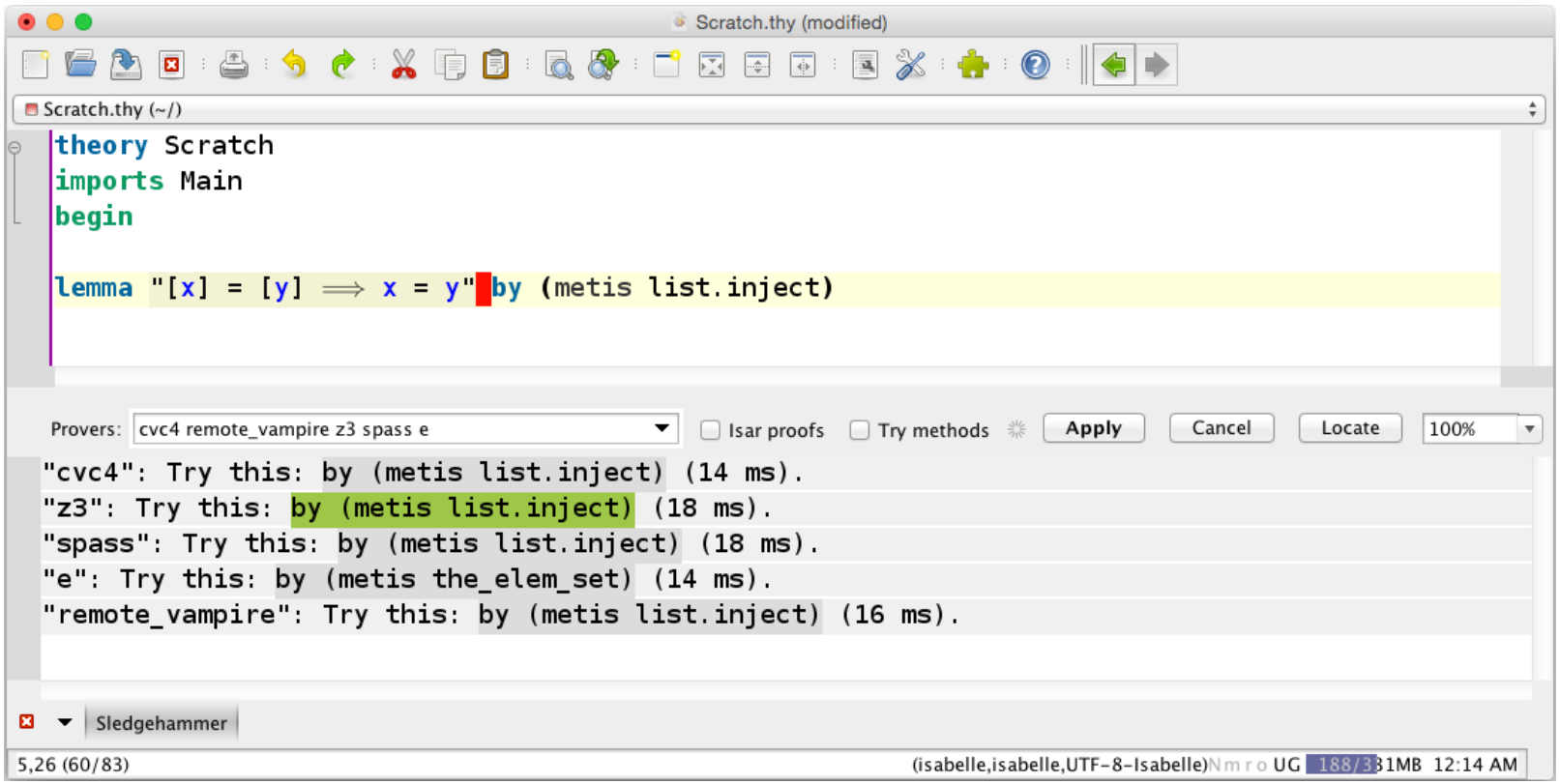
$$\texttt{foldr}\ f\ b\ [x_1, x_2, \ldots, x_n] = f\ x_1\ (f\ x_2\ (\ldots (f\ x_n\ b)\ldots))$$

- How to prove correctness? $\Longrightarrow$ Isabelle/HOL

Isabelle/HOL proof tools

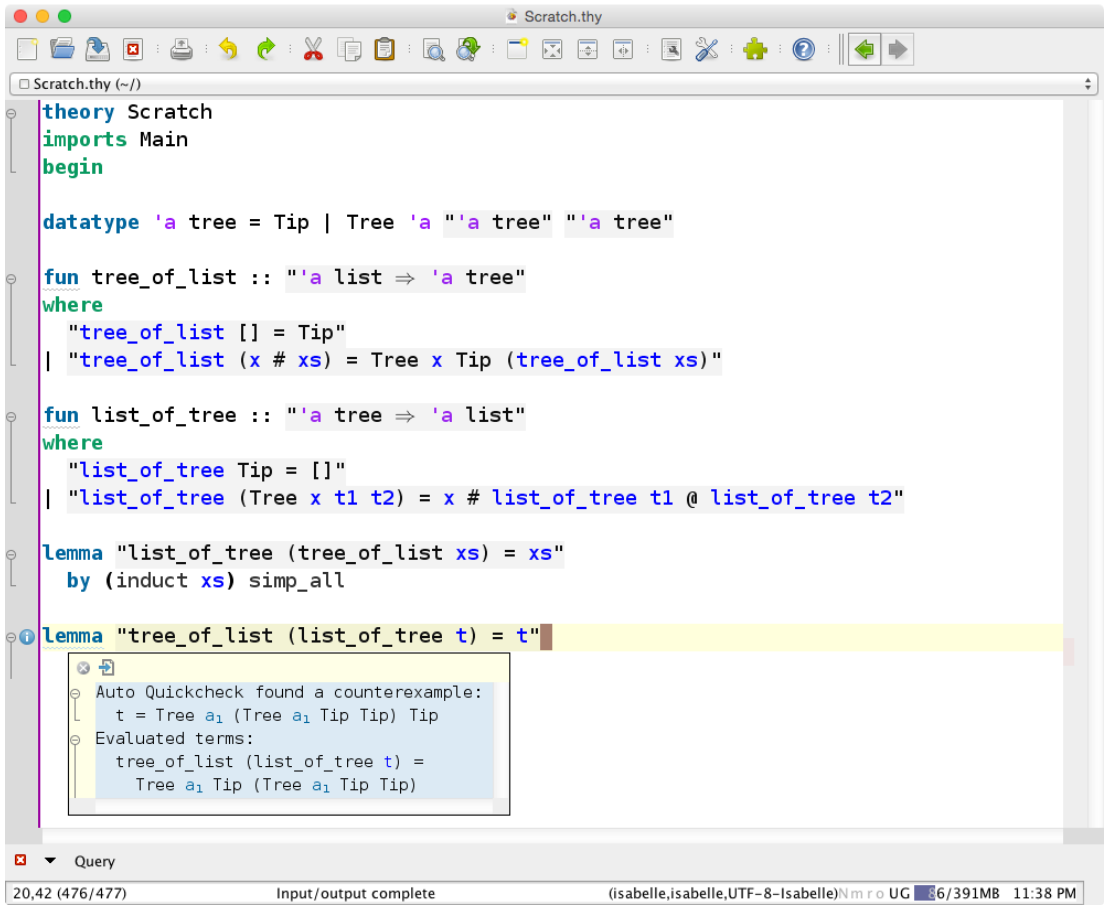
Sledgehammer

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



Automated disprovers — counter examples

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



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 $'a\ list = Nil \mid Cons\ (hd\colon 'a)\ (tl\colon 'a\ list)$
codatatype $'a\ stream = Stream\ (HD\colon 'a)\ (TL\colon 'a\ stream)$

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
$base$: $(x,\ x) \in R\star$
| $step$: $(x,\ y) \in R \Longrightarrow (y,\ z) \in R\star \Longrightarrow (x,\ z) \in R\star$

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:

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

Type classes

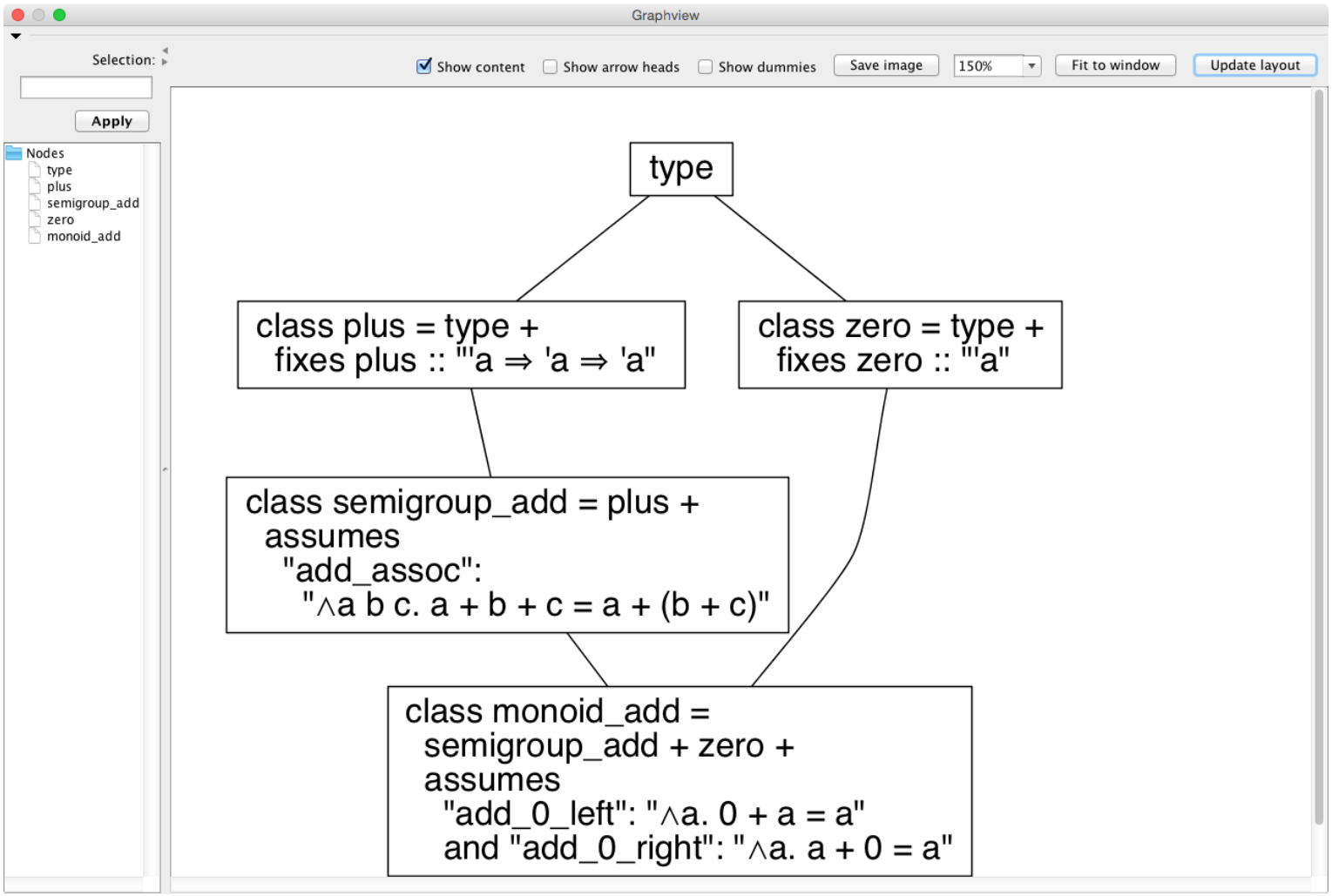
- Predicate over constant signature with single type-variable
- Integrated into type-system: order-sorted algebra of constraints
- Class intersections are called sorts
- Class inclusion hierarchy: by definition or proof
- Class instantiation by concrete types

class $zero = $ **fixes** $zero :: 'a \ \ (\mathbf{0})$
class $one = $ **fixes** $one \ :: 'a \ \ (\mathbf{1})$
class $times = $ **fixes** $times :: 'a \Rightarrow 'a \Rightarrow 'a \ \ (\textbf{infixl} * 70)$

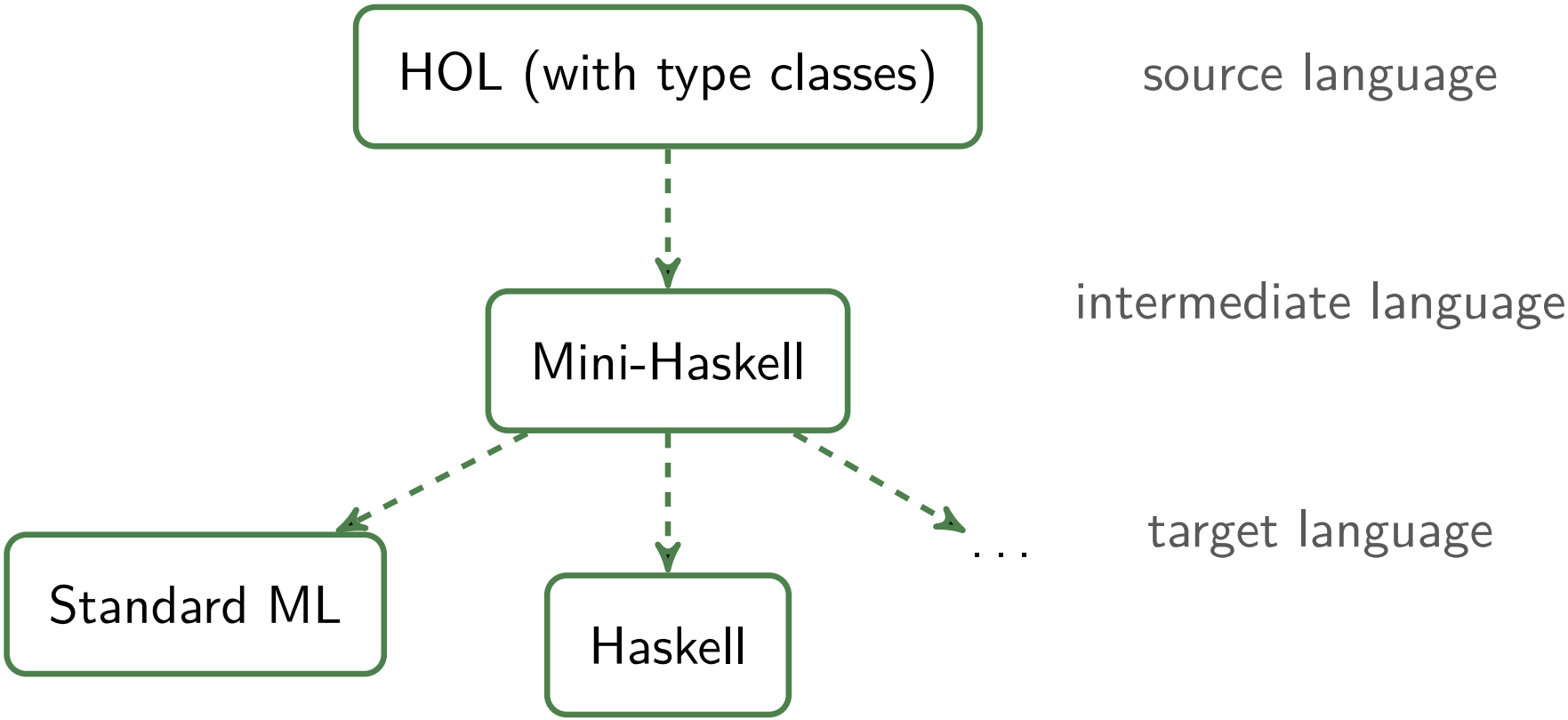
class $group = times + one + inverse +$
  assumes $group\_assoc$: $(x * y) * z = x * (y * z)$
    and $group\_left\_one$: $\mathbf{1} * x = x$
    and $group\_left\_inverse$: $inverse\ x * x = \mathbf{1}$

Example: class hierarchy

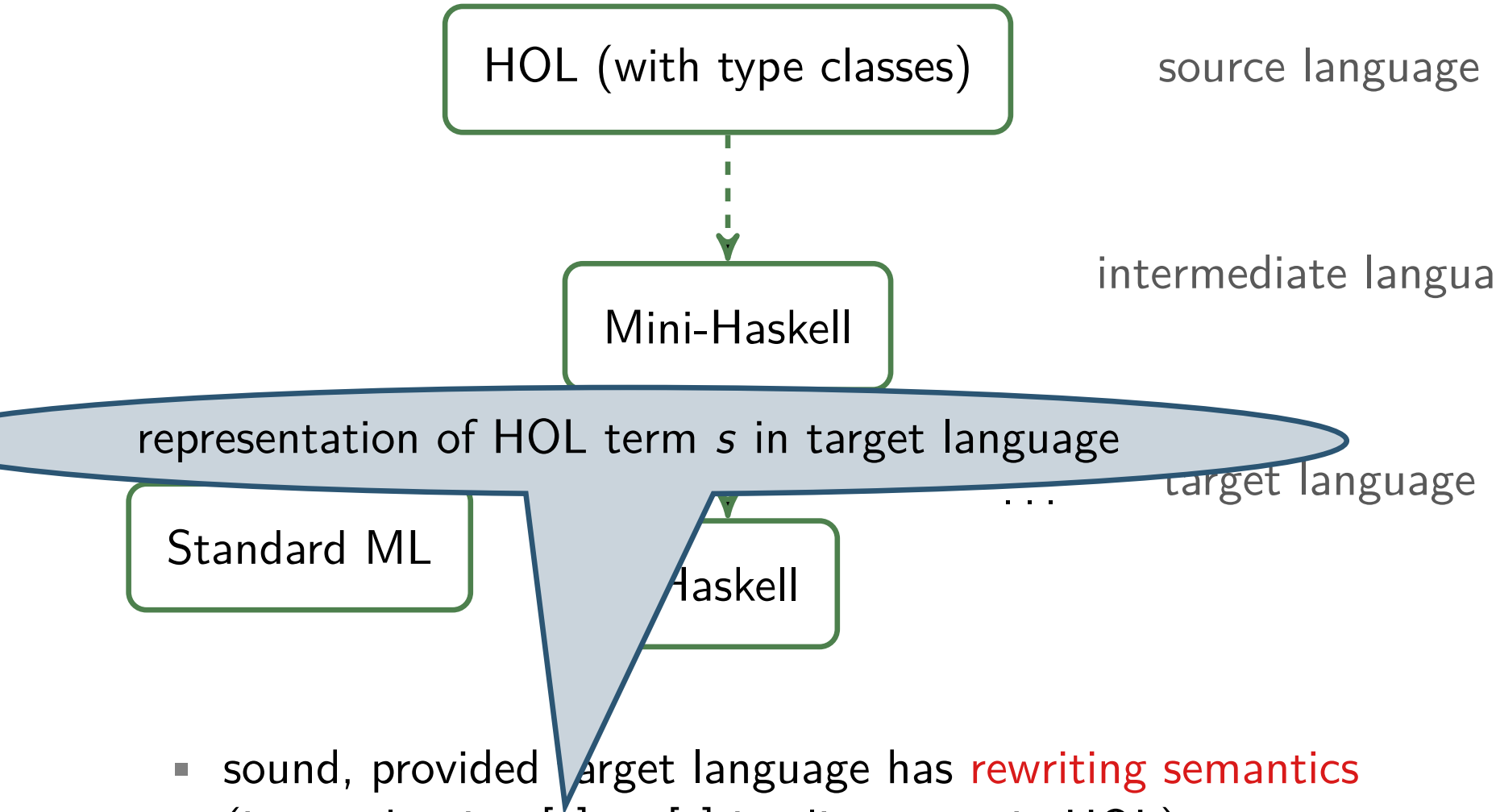
class_deps *type monoid_add*



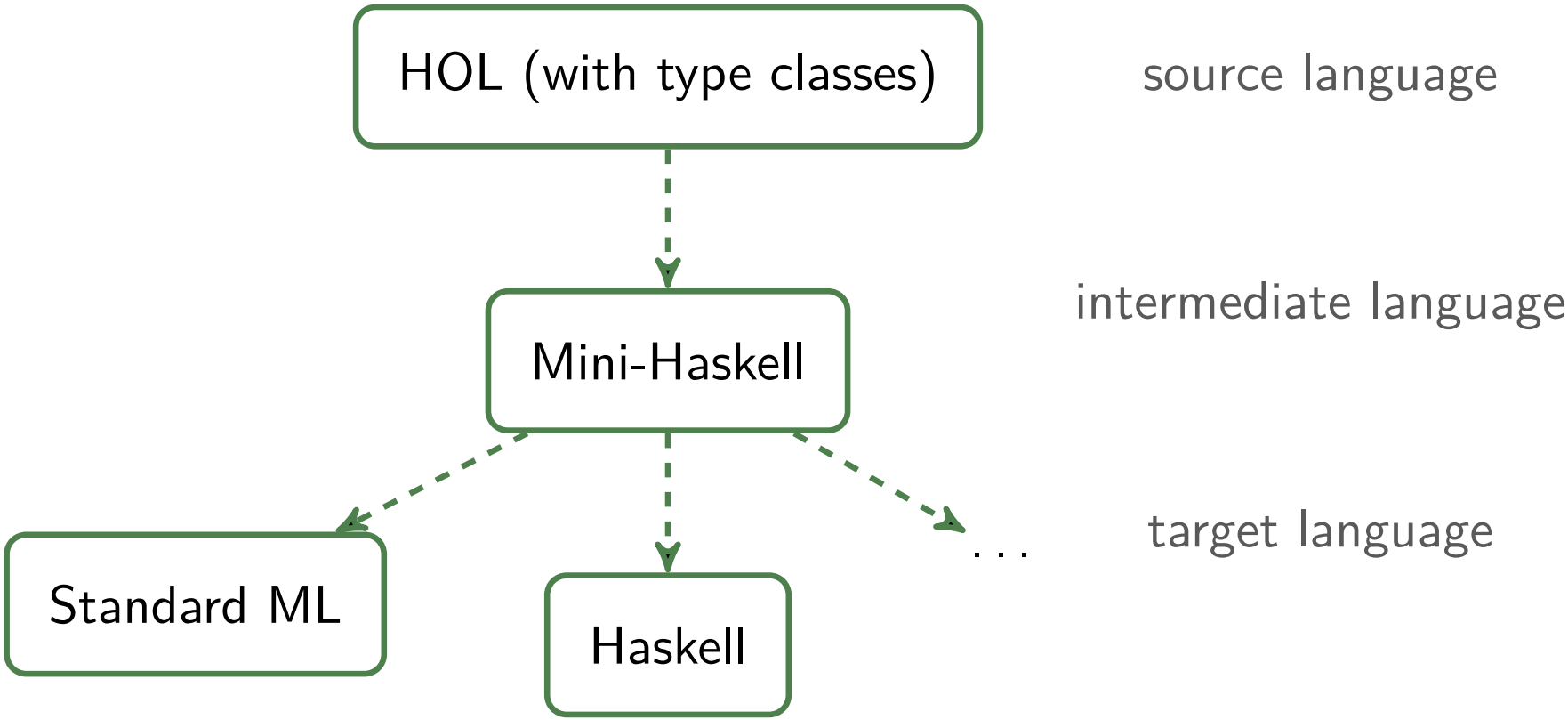
Code generation



- 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 $facts_1$ **have** $prop$ **using** $facts_2$
  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 $this$	$\equiv$	**then**
from a	$\equiv$	**note** a **then**
with a	$\equiv$	**note** a **and** $this$ **then**

Example

notepad
begin
  fix $A\ B :: bool$
  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**

{
  fix x
  have $B\ x$ **sorry**
}
have $\bigwedge x.\ B\ x$ **by** $fact$

{
  assume A
  have B **sorry**
}
have $A \Longrightarrow B$ **by** $fact$

- Existential context: **obtain**

{
  obtain a **where** $B\ a$ **sorry**
  have C **sorry**
}
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**