

## Chapter 6

# Inductive Definitions and Fixed Points

# Overview of Chapter

## 6. Inductive Definitions and Fixed Points

6.1 Inductively defined sets and predicates

6.2 Fixed point theory for inductive definitions

6.3 Specifying and verifying transition systems

# Introduction

## Constructs for defining types and functions

Isabelle/HOL provides two core constructs for conservative extensions:

1. Constant definitions
2. Type definitions

Based on the core construct, there are further constructs:

- Recursive function definitions (`primrec`, `fun`, `function`)
- Recursive datatype definitions (`datatype`)
- Co-/inductively defined sets (`inductive_set`, `coinductive_set`)
- Co-/inductively defined predicates (`inductive`, `coinductive`)

# Motivation

## Goals

- Learn about inductive definitions:
  - ↪ important concept in computer science!
  - E.g., to define operational semantics.
- Learn the underlying fixed point theory:
  - ↪ fundamental theory in computer science!
- Learn how to apply it to transition systems
  - ↪ central modeling concept for operational behavior!

# Section 6.1

## **Inductively defined sets and predicates**

# Introductory example

Informally:

- 0 is even
- If  $n$  is even, so is  $n + 2$
- These are the only even numbers

In Isabelle/HOL:

```
-- The set of all even numbers
inductive_set even :: "nat set" where
zero [intro!]    "0 ∈ even" |
step [intro!]    "n ∈ even  $\implies$  n + 2 ∈ even"
```

# Format of inductive definitions

`inductive_set S :: " $\alpha$  set" where`  
`"[[  $a_1 \in S; \dots; a_n \in S; A_1; \dots; A_k$ ]]  $\implies a \in S$ " |`  
`... |`  
`...`

where

- $A_1, \dots, A_k$  are side conditions not involving  $S$  and
- $a$  is a term built from  $a_1, \dots, a_n$ .

The rules can be given names and attributes as seen in definition of *even*.

# Embedding inductive definitions into HOL

## Conservative theory extension

From an inductive definition, Isabelle

- generates a *definition* using a fixed point operator and
- proves theorems about it that can be used as proof rules

The theory underlying fixed point definitions is explained in Subsect. 6.2.



# Generated rules

## Rules

Generated rules include

- the introduction rules of the definition, e.g.,

$$0 \in \text{even} \quad (\text{even.zero})$$

$$n \in \text{even} \implies n + 2 \in \text{even} \quad (\text{even.step})$$

- an elimination rule for case analysis
- an induction rule

# Proving simple properties of inductive sets

## Example 1:

Lemma:  $4 \in \text{even}$

Proof:  $0 \in \text{even} \implies 2 \in \text{even} \implies 4 \in \text{even}$

## Discussion:

- Simple: Use `even.zero` and apply rule `even.step` finitely many times.
- Works because there is no free variable

# Proving properties of inductive sets

## Example 2:

Lemma:  $m \in \text{even} \implies \exists k. 2 * k = m$

Proof: Idea:

- For rules of the form  $a \in S$ : Show that property holds for  $a$
- For rules of the form  $[[ a_1 \in S; \dots; a_n \in S; \dots ]] \implies a_0 \in S$ : Show that assuming  $a_1 \in S; \dots; a_n \in S; \dots$  and property holds for terms  $a_1, \dots, a_n$ , it holds for term  $a_0$

Applied to *even*, we have to show:

- $\exists k. 2 * k = 0$ : trivial
- Assuming  $n \in \text{even}$  and  $\exists k. 2 * k = n$ , show  $\exists k. 2 * k = n + 2$ : simple arithmetic

# Rule induction for *even*

To prove  $n \in \text{even} \implies P\ n$  by rule induction, one has to show:

- $P\ 0$
- $P\ n \implies P\ (n + 2)$

Isabelle provides the rule `even.induct`:

$$\llbracket n \in \text{even}; P\ 0; \bigwedge n. P\ n \implies P\ (n + 2) \rrbracket \implies P\ n$$

# Rule induction vs. natural/structural induction

## Remarks:

- Rule induction uses the induction steps of the inductive definition and not of the underlying datatype! It differs from natural/structural induction.
- In the context of partial recursive functions, a similar proof technique is often called computational or fixed point induction.

# Rule induction in general

Let  $S$  be an inductively defined set.

To prove  $x \in S \implies P x$  by rule induction on  $x \in S$ , we must prove for every rule:

$$\llbracket a_1 \in S; \dots; a_n \in S \rrbracket \implies a \in S$$

that  $P$  is preserved:

$$\llbracket P a_1; \dots; P a_n \rrbracket \implies P a$$

In Isabelle/HOL: `apply (induct rule: S.induct)`

# Inductive predicates

Isabelle/HOL also supports the inductive definition of predicates:

$$X \in S \quad \rightsquigarrow \quad S \ x$$

Example:

```
inductive even :: "nat  $\Rightarrow$  bool" where  
  "even 0" |  
  "even n  $\implies$  even (n+2)"
```

Comparison:

- predicate: simpler syntax
- set: direct usage of set operation, like  $\cup$ , etc.

Inductive predicates can be of type  $\alpha_1 \Rightarrow \dots \Rightarrow \alpha_n \Rightarrow \text{bool}$

# Further aspects

- Rule inversion and inductive cases (see IHT 7.1.5)
- Mutual inductive definitions (see IHT 7.1.6)
- Parameters in inductive definitions (see IHT 7.2)



## Section 6.2

# Fixed point theory for inductive definitions

# Motivation

## Introduction:

Inductive definitions can be considered as:

- Constant definition: define exactly one set (*semantic interpretation*)
- Axiom system: except all sets that satisfy the rules (*axiomatic interpretation*)
- Derivation system: show that an element is in a set by applying the rules (*derivational interpretation*)

Isabelle/HOL is based on the semantic interpretation. In addition, it allows to use the rules as part of the derivation system.

## Remark

The interpretations have advantages and disadvantages/problems.

# Illustrating the problems

## Problem of semantic interpretation:

We have to assign a set to any well-formed inductive definition.

## Example:

Which set should be assigned to `fooset`:

```
inductive_set fooset :: "nat set" where  
  "n ∈ fooset ⇒ n+1 ∈ fooset"
```

## Problem of derivational interpretation

The rules of the definition are too weak. E.g., we cannot prove:

$$3 \notin \text{even}$$

# “Looseness” of rules

## Problem of axiomatic interpretation:

There are usually many sets satisfying the rules of an inductive definition.

## Example:

The following set `even2` satisfies the rules of `even`:

```
definition even2 :: "nat set" where
  "even2 ≡ { n. n ≠ 1 }"
```

```
lemma "0 ∈ even2"
```

```
lemma "n ∈ even2 ⇒ n+2 ∈ even2"
```

# Semantics of inductive definition

## Definition

Let  $f :: T \Rightarrow T$  be a function. A value  $x$  is called a *fixed point* of  $f$  if  $x = f x$ .

## Semantics approach for inductive definitions

Three steps:

- Transform inductive definition  $ID$  into “normalized form”
- “Extract” a fixed point equation for a function  $F_{ID} :: \alpha \text{ set} \Rightarrow \alpha \text{ set}$
- Take the least fixed point

## Assumption

For every (well-formed) inductive definition, the least fixed point exists.

## Transformation to “normalized form”

A “normalized” inductive definition has exactly one implication of the form:

```
inductive_set S :: "α set" where
  "m ∈ (F_S S) ⇒ m ∈ S"
```

Example:

```
inductive_set even :: "nat set" where
  "0 ∈ even" |
  "n ∈ even ⇒ n+2 ∈ even"
```

has the normalized form:

```
inductive_set even :: "nat set" where
  "m ∈ {m. m=0 ∨ (∃n. n ∈ even ∧ m=n+2)} ⇒ m ∈ even"
```

That is, the function  $F_{\text{even}}$  is

$$F_{\text{even}} \text{ nset} = \{m. m=0 \vee (\exists n. n \in \text{nset} \wedge m=n+2)\}$$

# Fixed point equation and existence of fixed points

Fixed point equation for a “normalized” inductive definition:

$$F_S \ S = S$$

## Existence of fixed points:

Unique least and greatest fixed points exist if

1.  $F_S$  is monotone, i.e.,  $F_S \ S \subseteq S$  for all  $S$ .
2. Domain (and range) of  $F_S$  is a complete lattice (Knaster-Tarski theorem)

Prerequisites are satisfied for inductive definitions, because

1. In inductive definitions, occurrence of  $x \in S$  must be *positive*, and this allows to prove monotonicity.
2. Set of sets are a complete lattice with  $\subseteq$  as ordering.

# Supremum and infimum

## Definition (Supremum/infimum)

Let  $(L, \leq)$  be partially ordered set and  $A \subseteq L$ .

- **Supremum:**  $y \in L$  is called a *supremum* of  $A$  if  $y$  is an upper bound of  $A$ , i.e.,  $b \leq y$  for all  $b \in A$  and

$$\forall y' \in L : ((y' \text{ upper bound of } A) \longrightarrow y \leq y')$$

- **Infimum:** analogously defined, greatest lower bound



# Complete lattices

## Definition (Complete lattice)

A partially ordered set  $(L, \leq)$  is a **complete lattice** if every subset  $A$  of  $L$  has both an infimum (also called the meet) and a supremum (also called the join) in  $L$ .

The meet is denoted by  $\bigwedge A$ , the join by  $\bigvee A$ .

## Lemma

*Complete lattices are non empty.*

## Lemma

*Let  $\mathcal{P}(S)$  be the power set of a set  $S$ .  
 $(\mathcal{P}(S), \subseteq)$  is a complete lattice.*

# Existence and structure of fixed points

## Theorem (Knaster-Tarski)

*Let  $(L, \leq)$  be a complete lattice and let  $F : L \rightarrow L$  be a monotone function. Then the set of fixed points of  $F$  in  $L$  is also a complete lattice.*

## Corollary (Knaster-Tarski)

$F$  has a (unique) least and greatest fixed point.

# Proof of Knaster-Tarski Corollary

We prove:

The set of all fixed points  $P$  of  $F$ ,  $P \subseteq L$ , has the following properties:

1.  $\bigvee P = \bigvee \{ y \in L \mid y \leq F(y) \}$
2.  $(\bigvee P) \in P$
3.  $\bigwedge P = \bigwedge \{ y \in L \mid F(y) \leq y \}$
4.  $(\bigwedge P) \in P$

That is,  $(\bigvee P)$  is the greatest and  $(\bigwedge P) \in P$  the least fixed point.

**Proof:**

We show the first two properties. The proof of the third and fourth property are analogous.

## Proof of Knaster-Tarski Corollary (2)

**Show:**  $\bigvee P = \bigvee \{ y \in L \mid y \leq F(y) \}$  and  $(\bigvee P) \in P$

Let  $D = \{ y \in L \mid y \leq F(y) \}$  and  $u = \bigvee D$ . We show:

$u \in P$  and  $u = \bigvee P$ , i.e.,  $u$  is the greatest fixed point of  $F$ .

For all  $x \in D$ , also  $F(x) \in D$ , because  $F$  is monotone and  $F(x) \leq F(F(x))$ .

$F(u)$  is an upper bound of  $D$ , because for  $x \in D$ ,  $x \leq u$  and  $F(x) \leq F(u)$ , i.e.,  $x \leq F(x) \leq F(u)$ .

As  $u$  is least upper bound,  $u \leq F(u)$ . Thus,  $u \in D$ .

As shown above,  $u \in D$  implies  $F(u) \in D$ , thus  $F(u) \leq u$ .

In summary,  $F(u) = u$ , i.e.,  $u$  is a fixed point,  $u \in P$ .

Because  $P \subseteq D$ ,  $\bigvee P \leq \bigvee D$ , hence  $u \leq \bigvee P \leq u$ , i.e.,  $u = \bigvee P$ .

# Lattices in Isabelle/HOL

## Remark

Isabelle/HOL handles:

- lattices in Chapter 5 of theory Main
- complete lattices in Chapter 8 of theory Main
- inductive definitions and Knaster-Tarski in Chapter 9

The natural numbers are introduced in Chapter 15, using an inductive definition!

# Some related definitions and lemmas in Isabelle/HOL

$mono\ f \equiv \forall A\ B. A \leq B \longrightarrow f\ A \leq f\ B$  (mono\_def)

where  $A, B$  are often sets and “ $\leq$ ” is “ $\subseteq$ ”

$lfp\ f \equiv Inf\ \{u \mid f\ u \leq u\}$  (lfp\_def)

$mono\ f \Longrightarrow lfp\ f = f\ (lfp\ f)$  (lfp\_unfold)

$\llbracket mono\ f; f\ (inf\ (lfp\ f)\ P) \leq P \rrbracket \Longrightarrow lfp\ f \leq P$  (lfp\_induct)

$gfp\ f \equiv Sup\ \{u \mid u \leq f\ u\}$  (gfp\_def)

$mono\ f \Longrightarrow gfx\ f = f\ (gfx\ f)$  (gfx\_unfold)

$\llbracket mono\ f; X \leq f\ (sup\ X\ (gfx\ f)) \rrbracket \Longrightarrow X \leq gfx\ f$  (coinduct)

## Section 6.3

# Specifying and verifying transition systems

# Motivation

## Modeling

Behavior of software-controlled systems can be modeled

- by using a modeling language (UML, B, Z, ASM, ABS, Maude, ...)
- by formalizing the operational behavior as transition system

## Transition systems

Transition systems are also a fundamental means for specifying

- the operational semantics of programming and modeling language (cf. Chap. 7)
- process calculi and concurrency
- computing architectures and hardware

Verification of transition systems cannot exploit program structure, but need other techniques.



# Transition systems

## Definition (Transition system)

A **transition system** (TS) is a pair  $(Q, T)$  consisting of

- a set  $Q$  of states;
- a binary relation  $T \subseteq Q \times Q$ , usually called the *transition relation*.

Notation:  $q \longrightarrow q'$

(Other names: state transition system, unlabeled transition system)

## Definition (Labeled transition system)

A **labeled transition system** (LTS) over  $Act$  is a pair  $(Q, T)$  consisting of

- a set  $Q$  of states;
- a ternary relation  $T \subseteq Q \times Act \times Q$ , usually called the transition relation. Notation:  $q \xrightarrow{lab} q'$ ,  $lab \in Act$

$Act$  is called the set of **actions** or **labels**.

# Transition systems (2)

## Remark

- The action labels express input, output, or an “explanation” of an internal state change.
- Finite automata are LTS.
- Often, transition systems are equipped with a set of initial states or sets of initial and final states.
- **Traces** are sequences  $\langle q_i \rangle$  of states with  $(q_i, q_{i+1}) \in T$  or sequences of labels
- **Behaviors** are sets of traces (beginning at initial states)
- **Properties** are often expressed in appropriate logics (PDL, CTL ...)

## Transition systems (3)

### Lemma

Every LTS  $(Q, T)$  over  $Act$  can be expressed by a TS  $(Q', T')$  such that there is a mapping

$$rep :: Q \times Act \Rightarrow Q'$$

with

$$q_1 \xrightarrow{lab} q_2 \in T \iff \exists lab. rep(q_1, lab) \longrightarrow rep(q_2, lab) \in T'$$

(Proof is a left as an exercise)

# Modeling: Case study Elevator control system

## Requirements

Design the control for an elevator serving 3 floors such that:

- Model:
  - ▶ Elevator has for each floor one button which, if pressed, causes it to visit that floor. Button is cancelled when the elevator visits the floor.
  - ▶ Each floor has a button to request the elevator. Button is cancelled when elevator visits the floor.
  - ▶ The elevator remains in the middle floor if no requests are pending.
- Properties:
  - ▶ All requests for floors from the elevator must be serviced eventually.
  - ▶ All requests from floors must be serviced eventually.

# Modeling approach and motivation

- Direct modeling as a transition system:
  - ▶ without using a programming or modeling language
  - ▶ without using a library/theory
- Motivation:
  - ▶ Learn to construct models
  - ▶ Deepen the knowledge about transition systems
  - ▶ Understand the formalization of transition systems

# Datatypes for facts and actions

```
datatype floor = F0 | F1 | F2           (* three floors *)

datatype action = Call floor           (* input message *)
               | GoTo floor           (* input message *)
               | Open                  (* output message *)
               | Move                  (* internal message *)

datatype direction = UP | DW           (* up | down *)
datatype door      = CL | OP           (* closed | open *)

type_synonym state =
  action × floor × direction × door × (floor set)
(* what , where , where to , door state , requests *)
```

# Datatypes and actions: Transition relation

```

inductive_set tr :: (state × state) set where
  [[ g ∉ T; ¬ (f = g ∧ d = OP) ]] ⇒
    ( (a, f, r, d, T), (Call g, f, r, d, T ∪ {g}) ) ∈ tr |
  [[ g ∉ T; ¬ (f = g ∧ d = OP) ]] ⇒
    ( (a, f, r, d, T), (GoTo g, f, r, d, T ∪ {g}) ) ∈ tr |
  f ∈ T ⇒ ( (a, f, r, d, T), (Open, f, r, OP, T - {f}) ) ∈ tr |
  ( (a, F1, r, d, {F0}), (Move, F0, DW, CL, {F0}) ) ∈ tr |
  ( (a, F1, r, d, {F2}), (Move, F2, UP, CL, {F2}) ) ∈ tr |
  F0 ∉ T ⇒ ( (a, F0, r, d, T), (Move, F1, UP, CL, T) ) ∈ tr |
  F2 ∉ T ⇒ ( (a, F2, r, d, T), (Move, F1, DW, CL, T) ) ∈ tr |
  [[ F1 ∉ T; F2 ∈ T ]] ⇒
    ( (a, F1, UP, d, T), (Move, F2, UP, CL, T) ) ∈ tr |
  [[ F1 ∉ T; F0 ∈ T ]] ⇒
    ( (a, F1, DW, d, T), (Move, F0, DW, CL, T) ) ∈ tr

```

# Traces

## Defining sets of infinite traces

```
types trace = "nat  $\Rightarrow$  state"
```

```
coinductive_set traces :: "trace set" where  
"[[ t  $\in$  traces; (s, t 0)  $\in$  tr ]]  $\implies$   
( $\lambda$ n. case n of 0  $\implies$  s | Suc x  $\implies$  t x)  $\in$  traces"
```

```
(* Functions on traces *)
```

```
definition head :: "trace  $\Rightarrow$  state" where  
"head t  $\equiv$  t 0"
```

```
definition drp :: "trace  $\Rightarrow$  nat  $\Rightarrow$  trace" where  
"drp t n  $\equiv$  ( $\lambda$  m. t (n + m))"
```



# Basic properties of traces

- lemma [iff]: `"drp (drp t n) m = drp t (n + m)"`
- lemma drp\_traces: `"t ∈ traces ⇒ drp t n ∈ traces"`

# More interesting properties

## Expressing temporal properties of traces

- For every floor  $f$ : If  $f$  is a requested floor, the elevator will eventually reach the floor and open the door in  $f$ :

**Always** ( $\ll\text{To } f\gg \rightarrow \text{Finally } (\ll\text{Op}\gg \text{ and } \ll\text{At } f\gg)$ )

Could be directly expressed over traces

- Alternative: Temporal logic, e.g., linear TL:
  - ▶ Formulas built with *Atoms*,  $\neg$ ,  $\wedge$ ,  $\square$ ,  $\diamond$
  - ▶ Interpretations: **Kripke structures**  $(Q, I, T, L)$
  - ▶ A transition relation  $T \subseteq Q \times Q$  such that  $\forall q \in Q. \exists q' \in Q. (q, q') \in T$
  - ▶ A labeling (or interpretation) function  $L :: Q \Rightarrow \mathcal{P}(\text{Atoms})$

# Syntax for LTL

## LTL formulas:

```
datatype formula = Atom atom           ("<< _ >>")
                 | Neg formula         ("¬")
                 | And formula formula (infixr ".^" 80)
                 | Always formula     ("□")
                 | Finally formula    ("◇")
```

## As abbreviation:

```
definition Imp :: "formula ⇒ formula ⇒ formula"
           (infixr ".→" 80)
```

where

```
"a .→ b = ¬ (a .^ ¬b)"
```

# Semantics for LTL

## Definition (Kripke structure)

Let  $AP$  be a set of atomic propositions. A *Kripke structure* is a 4-tuple  $M = (Q, I, T, L)$  consisting of

- a finite set of states  $Q$
- a set of initial states  $I \subseteq Q$
- a relation  $T \subseteq Q \times Q$  such that  $\forall q \in Q \exists q' \in Q$  with  $(q, q') \in T$
- a labeling (or interpretation) function  $L :: Q \Rightarrow \mathcal{P}(Atoms)$

# Kripke structure of elevator example

- $Q$  as defined by type synonym “state” (*UNIV state*)
- $I$ : some suitable set of initial states
- $T$  as defined by  $tr$  (why is there always a successor state?), and
- define  $AP \equiv atom$  and  $L$  as follows:

```
datatype atom = Up | Op | At floor | To floor
```

```
fun L :: "state  $\Rightarrow$  atom set" where
  "L (_, g, dr, ds, fs) =
    { a . (dr=UP  $\wedge$  a=Up)  $\vee$  (ds=OP  $\wedge$  a=Op)
       $\vee$  (a=At g)  $\vee$  ( $\exists$  f $\in$ fs.(a=To f)) }"
```

# Remarks and example

## Remarks:

- Since  $T$  is left-total, it is always possible to construct an infinite path through the Kripke structure. A **deadlock state**  $qd$  can be expressed by a single outgoing edge back to  $qd$  itself.
- The labeling function  $L$  defines for each state  $q$  in  $Q$  the set  $L(s)$  of all atomic propositions that are valid in  $s$ .
- Kripke structures are used to define the semantics of LTL (see next slide)

## Example of formalized property:

```
definition liveness :: "floor  $\Rightarrow$  formula" where
  "liveness f =  $\square$  ( $\llcorner$ To f $\gg$   $\cdot$   $\longrightarrow$   $\diamond$  ( $\llcorner$ Op $\gg$   $\cdot$   $\wedge$   $\llcorner$ At f $\gg$ ))"
```

# Semantics for LTL

Let  $M = (Q, I, T, L)$  be a *Kripke structure* and `trace` the type of traces defined by  $T$ :

```
primrec valid_in_trace ::
  "trace  $\Rightarrow$  formula  $\Rightarrow$  bool" ("(_  $\vDash$  _)" [80, 80] 80) where
  "t  $\vDash$   $\langle\langle a \rangle\rangle$  = ( a  $\in$  L (head t) )"
| "t  $\vDash$   $\neg$ f = (  $\neg$  (t  $\vDash$  f) )"
| "t  $\vDash$  f. $\wedge$  g = ( (t  $\vDash$  f)  $\wedge$  (t  $\vDash$  g) )"
| "t  $\vDash$   $\square$  f = (  $\forall$  n. ((drp t n)  $\vDash$  f) )"
| "t  $\vDash$   $\diamond$  f = (  $\exists$  n. ((drp t n)  $\vDash$  f) )"
```

```
definition valid :: "formula  $\Rightarrow$  bool"
  ("( $\vDash$  _)" [80] 80) where
  " $\vDash$  f  $\equiv$  ( $\forall$  t  $\in$  traces. t  $\vDash$  f)"
```

# Reasoning about finite transition systems

## Three options for reasoning:

1. In Isabelle/HOL using the rules obtained from the definitions (semantics-based, formalized mathematical reasoning):  
» [Elevator.thy](#) (see exercises)
2. In LTL using rules for temporal reasoning (rules not shown here)
3. Model checking (works for finite state systems)