Chapter 5

Verifying Functions

Overview of Chapter

Section 5.1

Introduction

Motivation

Verifying properties of functions

Verifying properties of functions is a fundamental task in theorem proving and software engineering:

- Functions allow to express recursive algorithms
- Functions can be used to model systems (e.g., a compiler is essentially a function)
- Functions are used to specify input/output behavior of procedures, so called IO-properties
- Verifying recursive functions is related to terminiation proofs

Specification

Kinds of specifications:

- specification = model + properties
  $\Longrightarrow$ verify that model has the properties

or

- specification = model_1 + model_2 + relationship
  $\Longrightarrow$ verify that models are in the relationship

Here:

specification = function definition + property of function

Basic proof techniques

Verify:

- well-definedness of function by:
  - structural induction according to parameter types
  - more general: well-founded ordering on parameter space: “show that parameters get smaller”
- property of defined function:
  - structural induction according to parameter types
  - in general, proof technique depends on properties

Discussion

Verification

- checks for consistency of models and properties
  - models may not reflect what designer/programmer had in mind
  - properties may not reflect what designer/programmer had in mind
- works for the full parameter space (in contrast to testing)
- discovers also “pathological” problems
- uses redundancy to find errors
- helps to improve the descriptions

Formal verification avoids misunderstanding, allows using tools, and avoids errors in proofs.

Section 5.2

Case study: Greatest common devisor

Mathematical specification of `gcd`

Sepcification

The function `gcd` should have the following property:
For m, n with $m \geq 0$, $n \geq 0$, m, n not both zero, it holds:

$$gcd\ \ m\ \ n = \max \{\ k \mid k \text{ divides } m \text{ and } n\ \}$$

Algorithm and property

Function definition

```
fun gcd :: "nat ⇒ nat ⇒ nat" where
"gcd m 0 = m" |
"gcd m n = gcd n (m mod n)"
```

Property of function

```
theorem gcd_greatest:
"(k dvd m ∧ k dvd n ∧ (0<m ∨ 0<n)) ⟶ (k ≤ gcd m n)"
```

Proofs:

» Gcd.thy

Mathematical proof of gcd

Lemma:
For $m \geq 0$, $n > 0$ we have:

k divides m and $n \quad \Leftrightarrow \quad k$ divides n and k divides (m mod n)

Proof of gcd by structural induction:
We show:

a) `gcd` is correct for $n = 0$ and arbitrary m.

b) Induction hypothesis:
`gcd` is correct for all pairs (m, k) for arbitrary $k \leq n$ and m;

Show:
`gcd` is correct for all pairs $(m, n + 1)$ for arbitrary m.

Mathematical proof of gcd (2)

(a) Induction base:

```
  gcd m 0
=
  m
=
  max { k | k divides m }
=
  max { k | k divides m and 0 }
```

Mathematical proof of gcd (3)

(b) Induction step:

Assumptions: n is given.
For all pairs (m, k) with $k \leq n$ it holds: `gcd` is correct for (m, k)

Show: For all m it holds: `gcd` is correct for $(m, n+1)$!

```
  gcd m (n+1)
=   (* Declaration of gcd *)
  gcd (n+1) (m mod (n+1))
=   (* m mod (n+1) ≤ n and induction hypothesis *)
  max { k |  k divides (n+1) and (m mod (n+1)) }
=   (* Lemma *)
  max { k |  k divides m and (n+1)}
QED.
```

Section 5.3

Well-definedness of total recursive functions

Outline

Well-definedness proofs:

- Show that there exists a well-founded relation *wf* on the arguments
- Show that arguments in recursive calls are smaller w.r.t. *wf*

What we need:

- Well-founded relations and induction
- Relations: Relations are sets in Isabelle/HOL
- Sets

» Sections 6.1, 6.2, 6.4 of Isabelle/HOL Tutorial

Sets in HOL

Introduction

Sets in HOL differ from sets in set theory:

- All elements of a set have the same type, say α.
- Sets are typed: α *set*
- Only some values are sets in HOL.

Intersection, complement, difference

Sample deduction rules for intersection:

$$[\![c \in A; c \in B]\!] \Longrightarrow c \in A \cap B \qquad (IntI)$$
$$c \in A \cap B \Longrightarrow c \in A \qquad (IntD1)$$
$$c \in A \cap B \Longrightarrow c \in B \qquad (IntD2)$$

Set complement and difference:

$$(c \in -A) = (c \notin A) \qquad (Compl\_iff)$$
$$-(A \cup B) = -A \cap -B \qquad (Compl\_Un)$$
$$A \cap (B - A) = \{\} \qquad (Diff\_disjoint)$$
$$A \cup -A = UNIV \qquad (Compl\_partition)$$

Subsets, extensionality, equality

Subsets:

$$(\bigwedge x.x \in A \Longrightarrow x \in B) \Longrightarrow A \subseteq B \qquad (subsetI)$$
$$[\![A \subseteq B; c \in A]\!] \Longrightarrow c \in B \qquad (subsetD)$$
$$(A \cup B \subseteq C) = (A \subseteq C \wedge B \subseteq C) \qquad (Un\_subset\_iff)$$

Extensionality and equality of sets:

$$(\bigwedge x.(x \in A) = (x \in B)) \Longrightarrow A = B \qquad (set\_ext)$$
$$[\![A \subseteq B; B \subseteq A]\!] \Longrightarrow A = B \qquad (equalityI)$$
$$[\![A = B; [\![A \subseteq B; B \subseteq A]\!] \Longrightarrow P]\!] \Longrightarrow P \qquad (equalityE)$$

Set comprehension

Subsets:

$$(a \in \{x.\ P\ x\}) = P\ a \qquad (mem\_Collect\_eq)$$
$$\{x.\ x \in A\} = A \qquad (Collect\_mem\_eq)$$

Some simple facts:

```
lemma "{x. P x ∨ x ∈ A} = {x. P x} ∪ A"
lemma "{x. P x ⟶ Q x} = -{x. P x} ∪ {x. Q x}"
```

More convenient syntax, example:

$$\{p * q \mid p\ q.\ p \in prime \wedge q \in prime\}$$
$$= \{z.\ \exists p\ q.\ z = p * q \wedge p \in prime \wedge q \in prime\}$$

Binding operators

Universal and existential quantification:

$$(\bigwedge x.\ x \in A \Longrightarrow P\ x) \Longrightarrow \forall x \in A.\ P\ x \qquad (ballI)$$
$$[\![\forall x \in A.\ P\ x;\ x \in A]\!] \Longrightarrow P\ x \qquad (bspec)$$
$$[\![P\ x;\ x \in A]\!] \Longrightarrow \exists\ x \in A.\ P\ x \qquad (bexI)$$
$$[\![\exists x \in A.\ P\ x;\ \bigwedge x.\ [\![x \in A;\ P\ x]\!] \Longrightarrow Q]\!] \Longrightarrow Q \qquad (bexE)$$

Unions over parameterized sets, written $\bigcup x \in A.\ B\ x$. There is one basic law and two natural deduction rules:

$$(b \in (\bigcup x \in A.\ B\ x)) = (\exists\ x \in A.\ b \in B\ x) \qquad (UN\_iff)$$
$$[\![a \in A;\ b \in B\ a]\!] \Longrightarrow b \in (\bigcup x \in A.\ B\ x) \qquad (UN\_I)$$
$$[\![b \in (\bigcup x \in A.\ B\ x);\ \bigwedge x.\ [\![x \in A;\ b \in B\ x]\!] \Longrightarrow R]\!] \Longrightarrow R \qquad (UN\_E)$$

Relations in HOL

Introduction

A relation in Isabelle/HOL is a set of pairs.

Relations are often defined by

- composition
- closure of another relation
- inverse image of a relation w.r.t. a function

Relation basics

Identity and composition of relations:

$$Id \equiv \{\, p.\ \exists x.\ p = (x, x) \,\} \qquad (Id\_def)$$
$$r\ O\ s \equiv \{(x, z).\ \exists y.\ (x, y) \in s \wedge (y, z) \in r \,\} \qquad (rel\_comp\_def)$$
$$R\ O\ Id = R \qquad (R\_O\_Id)$$
$$[\![r' \subseteq r;\ s' \subseteq s]\!] \Longrightarrow r'\ O\ s' \subseteq r\ O\ s \qquad (rel\_comp\_mono)$$

The converse or inverse of a relation exchanges the operands:

$$((a, b) \in r^{-1}) = ((b, a) \in r) \qquad (converse\_iff)$$

Here is a typical lemma proved about converse and composition:

```
lemma converse_rel_comp: "(r O s)^-1 = s^-1 O r^-1"
```

Closures

Isabelle/HOL defines the reflexive transitive closure r^* of a relation as the *least solution/fixpoint* of the equation:

$$r^* = Id \cup (r\ O\ r^*) \qquad (rtrancl\_unfold)$$

Basic properties:

$$(a, a) \in r^* \qquad (rtrancl\_refl)$$
$$p \in r \Longrightarrow p \in r^* \qquad (r\_into\_rtrancl)$$
$$[\![(a, b) \in r^*;\ (b, c) \in r^*]\!] \Longrightarrow (a, c) \in r^* \qquad (rtrancl\_trans)$$

Inverse image

Let

- r of type $(\alpha \times \alpha)$ *set* and
- f be be a function of type $\beta \Rightarrow \alpha$

The *inverse image* of r w.r.t. to f is:

$$inv\_image\ r\ f \equiv \{(x, y).\ (f\ x,\ f\ y) \in r\} \qquad (inv\_image\_def)$$

Remark

Inverse images are helpful for defining a new well-founded relation from a known well-founded relation r.

Well-founded relations

Intuitively, a relation $\prec$ is *well-founded* if every descending chain of elements is finite; i.e., there is no infinite descending chain of elements $a_0, a_1 \ldots$:

$$\cdots \prec a_2 \prec a_1 \prec a_0$$

Isabelle/HOL provides a predicate *wf* that asserts that a relation is well-founded; e.g., for *less_than* :: $(nat \times nat)$ *set* :

$$((x, y) \in less\_than) = (x < y) \qquad (less\_than\_iff)$$
$$wf\ less\_than \qquad (wf\_less\_than)$$

Problem

It can be difficult to prove *wf r* for a relation *r*.

Proving well-foundedness

Proof method

To prove that a relation r is well-founded, show that it is an inverse image of a well-founded relation w.r.t. to some “measure” function f.

Example proof of well-foundedness

Let

```
definition shorter :: "('a list × 'a list) set" where
  "shorter =  { (xl,yl) . length xl < length yl }"
```

Lemma: `shorter` is well-founded, i.e., `"wf shorter"`

Proof: Show that `shorter` is inverse image of measure function `length`:

```
"shorter = inv_image less_than length"
```

Then, `"wf less_than"` and

```
theorem wf_inv_image: "wf r ⟹ wf (inv_image r f)"
```

imply `"wf shorter"`

Proving well-definedness of functions

Well-definedness

A recursively defined function is *well-defined* if the arguments in all recursive calls are smaller w.r.t. some well-founded relation.

Proving well-definedness

- Provide a so-called *measure* function f from the arguments to *nat*.
- Any such function defines a well-founded relation on the argument space:

$$\begin{array}{ll} measure \equiv inv\_image\ less\_than & (measure\_def) \\ wf\ (measure\ f) & (wf\_measure) \end{array}$$

- Show that the arguments of the recursive calls get smaller w.r.t. f.

Well-founded induction

Induction proofs based on well-founded relations

Well-founded relations r can be used for induction proofs:
A property holds for all elements iff we can show that it holds for an element x assuming it holds for all predecessors.

In Isabelle/HOL:

$$[\![\mathit{wf}\ r;\ \bigwedge x.\ \forall y.(\ (y, x) \in r \longrightarrow P\ y) \Longrightarrow\ P\ x\ ]\!] \Longrightarrow\ P\ a \qquad (\mathit{wf\_induct})$$

Remark

Note that in well-founded inductions, there is no explicit induction base.

Section 5.4

Case study: Quicksort

Analysing algorithms

Case study Quicksort

We analyse a functional version of the quicksort algorithm.

```
function qsort :: "('a::linorder) list ⇒ 'a list" where
  "qsort []     = []"
| "qsort (p#l) =       qsort (qsplit (op <) p l)
                  @ p # qsort (qsplit (op ≥) p l)"
```

where `linorder` is a type class supporting "<" and "≥" and

```
primrec qsplit :: "('a ⇒ 'a ⇒ bool) ⇒ 'a :: linorder ⇒
     'a list ⇒ 'a list" where
  "qsplit cr p [] = []"
| "qsplit cr p (h # t) =
   (if cr h p then h # qsplit cr p t else qsplit cr p t)"
```

Properties to prove

Properties:

1. Well-definedness of qsort
2. (Well-definedness of qsplit)
3. Sortedness of result
4. Result is a permutation of input list

Specifying sortedness

Sortedness:

```
fun qsorted :: "'a :: linorder list ⇒ bool" where
  "qsorted [] = True"
| "qsorted [x] = True"
| "qsorted (a # b # l) = (b ≥ a ∧ qsorted (b # l))"

lemma qsort_sorts: "qsorted (qsort xl)"
```

Specifying the permutation property

Permutation using a multiset abstraction:

```
primrec count :: "'a list ⇒ 'a ⇒ nat" where
  "count []       =  (λ x. 0)"
| "count (h # t) =  (count t) (h := count t h + 1)"

lemma qsort_preserves:  "count (qsort xl) = count xl"
```

Verification

The proofs for the properties are presented step by step in the lecture.

Resulting theory with proofs:

» `Quicksort.thy`