Verifying Functions

Section 5.1

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 termination proofs
5. Verifying Functions

5.1 Introduction

Specification

Kinds of specifications:

- specification = model + properties
  \[\Rightarrow\] verify that model has the properties

or

- specification = model₁ + model₂ + relationship
  \[\Rightarrow\] 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 divisor
5. Verifying Functions

5.2 Case study: Greatest common divisor

Mathematical specification of gcd

Specification
The function \( \text{gcd} \) should have the following property:
For \( m, n \) with \( m \geq 0, \ n \geq 0 \), \( m, n \) not both zero, it holds:

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

Algorithm and property

Function definition
fun \( \text{gcd} \) :: \( \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \) where
"\( \text{gcd} \ m \ 0 = m \)"
"\( \text{gcd} \ m \ n = \text{gcd} \ n \ (m \ mod \ n) \)"

Property of function
theorem \( \text{gcd}\_\text{greatest} \):
"(k dvd m \land k dvd n \land (0<m \lor 0<n)) \rightarrow (k \leq \text{gcd} \ m \ n)"

Proofs:
» Gcd.thy

Mathematical proof of \( \text{gcd} \)

Lemma:
For \( m \geq 0, \ n > 0 \) we have:
\( k \) divides \( m, n \) \( \iff \) \( k \) divides \( n \) and \( k \) divides \( (m \ mod \ n) \)

Proof of \( \text{gcd} \) by structural induction:
We show:

a) \( \text{gcd} \) is correct for \( n = 0 \) and arbitrary \( m \).

b) Induction hypothesis:
\( \text{gcd} \) is correct for all pairs \((m,k)\) for arbitrary \( k \leq n \) and \( m \);
Show:
\( \text{gcd} \) is correct for all pairs \((m,n+1)\) for arbitrary \( m \).
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 $(m, n + 1)$!

\[
gcd \cdot m \ (n+1) = (* \ Declaration \ of \ gcd \ *)
gcd \ (n+1) \ (m \ mod \ (n+1)) = (* m \ mod \ (n+1) \leq n \ and \ induction \ hypothesis \ *)
\max \ { k \ | \ k \ divides \ (n+1) \ and \ (m \ mod \ (n+1)) } = (* \ Lemma \ *)
\max \ { k \ | \ k \ divides \ m \ and \ (n+1)}
\]
QED.

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

Introduction
Sets in HOL differ from sets in set theory:
- All elements of a set have the same type, say $\alpha$.
- Sets are typed: $\alpha$ set
- Only some values are sets in HOL.
Set comprehension

Subsets:

\[ a \in \{ x . P x \} \] = \( P a \)  \hspace{1cm} (mem_Collect_eq)
\[ \{ x . x \in A \} = A \]  \hspace{1cm} (Collect_mem_eq)

Some simple facts:

\textbf{lemma} "[x. P x \lor x \in A] = [x. P x] \cup [x. A]"
\textbf{lemma} "[x. P x \rightarrow Q x] = \neg [x. P x] \cup [x. Q x]"

More convenient syntax, example:

\[ \{ p \ast q | p, q, p \in \text{prime} \land q \in \text{prime} \} = \{ z . \exists p, q. z = p \ast q \land p \in \text{prime} \land q \in \text{prime} \} \]

Intersection, complement, difference

Sample deduction rules for intersection:

\[ [c \in A; c \in B] \Longrightarrow c \in A \cap B \]  \hspace{1cm} (Intl)
\[ c \in A \cap B \Longrightarrow c \in A \]  \hspace{1cm} (IntD1)
\[ c \in A \cap B \Longrightarrow c \in B \]  \hspace{1cm} (IntD2)

Set complement and difference:

\[ (c \in \neg A) = (c \notin A) \]  \hspace{1cm} (Compl_iff)
\[ \neg (A \cup B) = \neg A \land \neg B \]  \hspace{1cm} (Compl_Un)
\[ A \cap (B - A) = \{ \} \]  \hspace{1cm} (Diff_disjoint)
\[ A \cup \neg A = \text{UNIV} \]  \hspace{1cm} (Compl_partition)

5. Verifying Functions
5.3 Well-definedness of total recursive functions

Subsets, extensionality, equality

Subsets:

\[ (\forall x . x \in A \Longrightarrow x \in B) \Longrightarrow A \subseteq B \]  \hspace{1cm} (subsetI)
\[ [A \subseteq B; c \in A] \Longrightarrow c \in B \]  \hspace{1cm} (subsetD)
\[ (A \cup B \subseteq C) = (A \subseteq C \land B \subseteq C) \]  \hspace{1cm} (Un_subset_if)

Extensionality and equality of sets:

\[ (\forall x . (x \in A) = (x \in B)) \Longrightarrow A = B \]  \hspace{1cm} (set_ext)
\[ [A \subseteq B; B \subseteq A] \Longrightarrow A = B \]  \hspace{1cm} (equalityI)
\[ [A = B; [A \subseteq B; B \subseteq A]] \Longrightarrow P \]  \hspace{1cm} (equalityE)

Binding operators

Universal and existential quantification:

\[ (\forall x . x \in A \Longrightarrow P x) \Longrightarrow \forall x \in A . P x \]  \hspace{1cm} (ball)
\[ [\forall x \in A . P x; x \in A] \Longrightarrow P x \]  \hspace{1cm} (bspec)
\[ [P x; x \in A] \Longrightarrow \exists x \in A . P x \]  \hspace{1cm} (bex1)
\[ \exists x \in A . P x; \forall x . [x \in A; P x] \Longrightarrow Q \]  \hspace{1cm} (bexE)

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

\[ (b \in (\bigcup x . A . B x)) = (\exists x . x \in A . b \in B x) \]  \hspace{1cm} (UN_iff)
\[ [a \in A; b \in B a] \Longrightarrow b \in (\bigcup x . A . B x) \]  \hspace{1cm} (UN_I)
\[ [b \in (\bigcup x . A . B x); \forall x . [x \in A; b \in B x] \Longrightarrow R] \Longrightarrow R \]  \hspace{1cm} (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

Closures

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

\[ r^* = \text{Id} \cup (r O r^*) \]

Basic properties:

- $(a, a) \in r^*$ \quad (\text{rtrancl_refl})
- $p \in r \Longrightarrow p \in r^*$ \quad (\text{r_into_rtrancl})
- $[(a, b) \in r'; (b, c) \in r^*] \Longrightarrow (a, c) \in r^*$ \quad (\text{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:

\[ \text{inv_image } r \ f \equiv \{(x, y). \ (f \ x, \ f \ y) \in r\} \]

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

$$\cdots < a_2 < a_1 < a_0$$

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

$$(x, y) \in \text{less\_than} = (x < y) \quad (\text{less\_than\_iff})$$

$\text{wf \_less\_than}$

Problem
It can be difficult to prove $\text{wf } r$ for a relation $r$.

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., "$\text{wf shorter}"

Proof: Show that shorter is inverse image of measure function length:
"shorter = inv_image less\_than length"

Then, "$\text{wf less\_than}$" and

theorem wf\_inv\_image: "$\text{wf } r \implies \text{wf } (\text{inv\_image } r \ f)$"

imply "$\text{wf shorter}"

Proving well-foundedness

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 $\text{nat}$.
• Any such function defines a well-founded relation on the argument space:

$$\text{measure} \equiv \text{inv\_image } \text{less\_than} \quad (\text{measure\_def})$$

$$\text{wf } (\text{measure } f) \quad (\text{wf\_measure})$$

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

© Arnd Poetzsch-Heffter et al. TUKaiserslautern 272
5. Verifying Functions

5.3 Well-definedness of total recursive functions

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:

\[
\begin{align*}
& \text{⟦ } \text{wf } r, \forall x. \forall y. (y, x) \in r \rightarrow P y \Rightarrow P x \text{ ⟧} \\
& \Rightarrow P a
\end{align*}
\]

(wf_induct)

Remark

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

5.4 Case study: Quicksort

Analysing algorithms

Case study Quicksort

We analyse a functional version of the quicksort algorithm.

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

where \texttt{linorder} is a type class supporting "<" and "\(\geq\)" and

```isabelle
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 \texttt{qsort}
2. (Well-definedness of \texttt{qsplit})
3. Sortedness of result
4. Result is a permutation of input list
Specifying the permutation property

Permutation using a multiset abstraction:

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)"

The proofs for the properties are presented step by step in the lecture.
Resulting theory with proofs:
» Quicksort.thy

Verifying Functions

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)"

 Verification

The proofs for the properties are presented step by step in the lecture.
Resulting theory with proofs:
» Quicksort.thy