Defining Functions on Equivalence Classes

Lawrence C. Paulson, Computer Laboratory,
University of Cambridge

Outline of Talk

1. Review of equivalence relations and quotients
2. General lemmas for defining quotients formally
3. Detailed development of the integers
4. Brief treatment of a quotiented datatype

Quotient Constructions

Identify values according to an equivalence relation

- terms that differ only by bound variable names
- numbers that leave the same residue modulo p

numerous applications in algebra, topology, *etc.*

- quotient constructions of the integers, rationals and non-standard reals; quotient groups and rings

Where are the applications in automated proof?

Definitions

- An *equivalence relation* $\sim$ on a set A is any relation that is reflexive (on A), symmetric and transitive.

- An *equivalence class* $[x]_\sim$ contains all y where $y \sim x$ (for $x \in A$)

- If $\sim$ is an equivalence relation on A, then the *quotient space* $A/\sim$ is the set of all equivalence classes

- The equivalence classes form a partition of A

Examples

- The integers: equivalence classes on $\mathbb{N} \times \mathbb{N}$

$$(x, y) \sim (u, v) \iff x + v = u + y$$

- The rationals: equivalence classes on $\mathbb{Z} \times \mathbb{Z}^{\neq 0}$

$$(x, y) \sim (u, v) \iff xv = uy$$

- λ-terms: equivalence classes on α-equivalence
- The hyperreals: infinite sequences of reals (quotiented with respect to an ultrafilter)

Constructing the Integers

$$[(x, y)] \quad \text{represents the integer} \quad x - y$$

The integer operations on equivalence classes:

$$\begin{aligned} 0 &= [(0, 0)] \\ -[(x, y)] &= [(y, x)] \\ [(x, y)] + [(u, v)] &= [(x + u, y + v)] \\ [(x, y)] \times [(u, v)] &= [(xu + yv, xv + vu)] \end{aligned}$$

Function definitions must preserve the equivalence relation. Then the choice of representative does not matter.

Sample Proof: $-(-z) = z$

- Replace z by an arbitrary equivalence class
- Rewrite using $-[(x, y)] = [(y, x)]$
- Proof is trivial:

$$-(-[(x, y)]) = -[(y, x)] = [(x, y)]$$

Proof that + is Associative

Replace each integer by a pair of natural numbers.

Prove by associativity of + on the naturals

$$\begin{aligned}\big([(x_1, y_1)] + [(x_2, y_2)]\big) + [(x_3, y_3)] &= [(x_1 + x_2 + x_3, y_1 + y_2 + y_3])\\ &= [(x_1, y_1)] + \big([(x_2, y_2)] + [(x_3, y_3)]\big)\end{aligned}$$

Alternatives to Quotients

- λ-terms? Use de Bruijn's treatment of variables ✓
- Integers as signed natural numbers? Ugly, with massive case analyses ✗
- Rationals as reduced fractions? Requires serious reasoning about greatest common divisors ✗
- Hyperreals? Quotient groups? ✗✗✗

Formalizing Quotients

Set comprehensions as nested unions of singletons

$$\{f(x_1, \ldots, x_n) \mid x_1 \in A_1, \ldots, x_n \in A_n\} = \bigcup_{x_1 \in A_1} \cdots \bigcup_{x_n \in A_n} \{f(x_1, \ldots, x_n)\}$$

Example: this definition of a quotient space

$$\text{"}A//r \equiv \bigcup x \in A.\ \{r``\{x\}\}\text{"}$$

The equivalence class $[x]$

Typical theorem: [x] = [y] if and only if $x \sim y$

```
theorem eq_equiv_class_iff:
  "⟦equiv A r;  x ∈ A;  y ∈ A⟧
   ⟹ (r``{x} = r``{y}) = ((x,y) ∈ r)"
```

r is an equivalence relation on A

The equivalence classes [x] and [y]

Defining Functions on Equivalence Classes

- Form a set by applying the concrete function to all representatives
- If the function preserves the equivalence relation, this set will be a singleton. Then get its element:

```
contents {x} = x
```

(Comprehensions are unions, so we collapse constant unions)

A Key Definition & Lemma

Congruence-preserving function, *f*:

```
congruent r f ≡ ∀y z. (y,z) ∈ r ⟶ f y = f z
```

Collapsing unions over equivalence classes,
where *f* is a set-valued function

```
lemma UN_equiv_class:
    "⟦equiv A r; congruent r f; a ∈ A⟧
     ⟹ (⋃x ∈ r``{a}. f x) = f a"
```

If *f* respects a equivalence relation, then the union over [*a*] is simply *f* (*a*).

Constructing the Integers

The equivalence relation:

```
intrel ≡ {((x,y),(u,v)) | x y u v. x+v = u+y}
```

The type definition (quotienting the universal set):

```
typedef (Integ) int = "UNIV//intrel"
  by (auto simp add: quotient_def)
```

The constants zero and one:

```
0 ≡ Abs_Integ(intrel ‘‘ {(0,0)})
1 ≡ Abs_Integ(intrel ‘‘ {(1,0)})
```

Defining Unary Minus

All representatives of the integer *z*

$$\texttt{-z} \equiv \texttt{contents}\ (\bigcup (x,y) \in \texttt{Rep\_Integ z}.\ \{\ \texttt{Abs\_Integ(intrel``\{(y,x)\})}\ \})$$

The equivalence class $[(y, x)]$

The desired *characteristic equation*: $-[(x, y)] = [(y, x)]$

Proving the Characteristic Equation

The definition respects the equivalence relation.

```
lemma minus:
  "- Abs_Integ(intrel``{(x,y)}) = Abs_Integ(intrel `` {(y,x)})"
proof -
  have "congruent intrel (λ(x,y).
    by (simp add: congruent_def)
  thus ?thesis
    by (simp add: minus_int_def UN_equiv_class [OF equiv_intrel])
qed
```

Result follows by definition, simplifying with a general lemma.

Reasoning About Minus

The characteristic equation lets other proofs resemble textbook ones.

Step 1: uses `cases` to replace each integer by an arbitrary pair of natural numbers.

Step 2: simplify using the equation and laws about the natural numbers.

```
lemma "- (- z) = z"
by (cases z, simp add: minus)
```

A Two-Argument Function

All representatives of the integers *z* and *w*

```
"z + w ≡
 contents (⋃(x,y)∈Rep_Integ z. ⋃(u,v)∈Rep_Integ w.
              { Abs_Integ(intrel``{(x+u, y+v)}) })"
```

The desired characteristic equation:

$$[(x, y)] + [(u, v)] = [(x + u, y + v)]$$

The obvious generalization of
the one-argument case

Proofs About Addition

The characteristic equation:

lemma *add:*

```
"Abs_Integ (intrel``{(x,y)}) + Abs_Integ (intrel``{(u,v)}) =
 Abs_Integ (intrel``{(x+u, y+v)})"
```

A typical theorem:

lemma *"-(z + w) = (-z) + (-w)"*
by *(cases z, cases w), simp add: minus add)*

Proof, as usual, by `cases` and simplification

Defining The Ordering

```
"z ≤ (w::int)
 ≡ ∃x y u v. x+v ≤ u+y &
               (x,y) ∈ Rep_Integ z & (u,v) ∈ Rep_Integ w"
```

The desired characteristic equation:

$$[(x, y)] \leq [(u, v)] \iff x + v \leq u + y$$

Its proof:

```
lemma le:
  "(Abs_Integ(intrel``{(x,y)}) ≤ Abs_Integ(intrel``{(u,v)}))
   = (x+v ≤ u+y)"
by (force simp add: le_int_def)
```

We are not forced to treat relations as functions.

How to Define a Quotiented Recursive Datatype

1. Define an ordinary datatype: a free algebra.
2. Define an equivalence relation expressing the desired equations.
3. Define the new type to be a quotient.
4. Define its abstract constructors and other operations as functions on equivalence classes.

A Message Datatype

```
datatype
      freemsg = NONCE  nat
              | MPAIR  freemsg freemsg
              | CRYPT  nat freemsg
              | DECRYPT  nat freemsg
```

Can encryption and decryption to be inverses?

$$D_K(E_K(X)) = X \text{ and } E_K(D_K(X)) = X$$

The Equivalence Relation

The desired equations

```
inductive "msgrel"
  intros
    CD:      "CRYPT K (DECRYPT K X) ∼ X"
    DC:      "DECRYPT K (CRYPT K X) ∼ X"
    NONCE:   "NONCE N ∼ NONCE N"
    MPAIR:   "⟦X ∼ X′; Y ∼ Y′⟧ ⟹ MPAIR X Y ∼ MPAIR X′ Y′"
    CRYPT:   "X ∼ X′ ⟹ CRYPT K X ∼ CRYPT K X′"
    DECRYPT: "X ∼ X′ ⟹ DECRYPT K X ∼ DECRYPT K X′"
    SYM:     "X ∼ Y ⟹ Y ∼ X"
    TRANS:   "⟦X ∼ Y; Y ∼ Z⟧ ⟹ X ∼ Z"
```

Symmetry and transitivity

For the abstract constructors

Defining Functions on the Quotiented Datatype

- Destructors: define first on the free datatype, respecting ~, then transfer.
- Constructors: define like other functions on equivalence relations. They respect ~ by its definition.

```
"Crypt K X == Abs_Msg (⋃U∈Rep_Msg X. msgrel``{CRYPT K U})"
```

Related Work

- HOL-4 packages by Harrison and Homeier
  - lift concrete functions to abstract ones
- Isabelle/HOL theories
  - Slotosch: partial equivalence relations
  - Wenzel: axiomatic type classes
- All using Axiom of Choice (Hilbert's ϵ-operator)

Conclusions

- Working with functions defined on quotient spaces is easy, using set comprehension.
- Any tool for set theory or HOL is suitable. (Arthan uses similar ideas with ProofPower.)
- The axiom of choice is not required.
- With correct lemmas, simplification is automatic.