Normal forms for monoid expressions

February 10, 2018

Contents

theory *MonoidNormalForms*
imports *Main ~~/src/HOL/Library/LaTeXsugar*
begin

This is a formalization of the monoid simplification we discussed in the exercises in the proof assistant Isabelle. Isabelle checks that steps in a proof are valid, so one no longer has to trust individual steps in a proof, just that the data types, functions and theorem statements make sense.

NOTE: this file is just here for people who are interested in how normal pen-and-paper proofs look like in proof assistants that mechanically check the correctness of proofs. Understanding Isabelle code is NOT required for the exam.

Isabelle code is somewhat similar to Haskell with a few differences: - : for lists is # - ++ is @ - type parameters are passed as prefix arguments to parametric types, e.g. int list instead of list int. The proof syntax resembles pen-and-paper proofs, so it should be readable without any exposure to Isabelle. You can ignore lines starting with by ¡something-long¿, which are generated by automated tools. "by simp" or "by auto" basically shows that a step is trivial enough for the basic automation of Isabelle to show them.

Note that the proofs here are much longer than they need to be to show the reasoning in more detail. Many steps could be solved by automated tools instead.

We assume that we have some type of variables:

typedecl *var*

for simplicity, we assume that var is countably infinite:

consts
$\mathcal{V}$ :: *var* $\Rightarrow$ *nat*
axiomatization where

var-countable: $\bigwedge x\ y.\ \mathcal{V}\ x = \mathcal{V}\ y \implies x = y$

This datatype encodes monoid expressions:

datatype *Monoid* = *One* | *Var var* | *Plus Monoid Monoid*

fun *simpl* :: *Monoid* ⇒ *var list* **where**
  simpl One = [] |
  simpl (*Var x*) = [*x*] |
  simpl (*Plus* e_1 e_2) = *simpl* e_1 @ *simpl* e_2

For simplicity, we model the Monoid type class using a record with two elements, and some nicer syntax for it.

record *'a monoid* =
  mult :: [*'a*, *'a*] ⇒ *'a* (**infixl** ⊗ı *70*)
  one :: *'a* (**1**ı)

Note that we need to write $\mathbf{1}_M$ for some monoid M to indicate which monoid's **1** we want to use. Similarly for ⊗

We also define what it to be a valid monoid instance:

definition *is-monoid* :: *'a monoid* ⇒ *bool* **where**
  is-monoid M ≡ ($\forall\ e.\ \mathbf{1}_M \otimes_M e = e \wedge e \otimes_M \mathbf{1}_M = e$) ∧
    ($\forall\ x\ y\ z.\ (x \otimes_M y) \otimes_M z = x \otimes_M (y \otimes_M z)$)

Evaluation then looks like in Haskell

fun *eval* :: *'a monoid* ⇒ (*var* ⇒ *'a*) ⇒ *Monoid* ⇒ *'a* **where**
  eval M env One = $\mathbf{1}_M$ |
  eval M env (*Plus a b*) = *eval M env a* $\otimes_M$ *eval M env b* |
  eval M env (*Var x*) = *env x*

We can also define evaluation for simplified expressions:

fun *eval'* :: *'a monoid* ⇒ (*var* ⇒ *'a*) ⇒ *var list* ⇒ *'a* **where**
  eval' M env [] = $\mathbf{1}_M$ |
  eval' M env (*x* # *xs*) = *env x* $\otimes_M$ *eval' M env xs*

A trivial helper lemma showing that appending lists in *eval'* and ⊗ commute; this follows immediately from induction on xs:

lemma *eval'-app*:
  assumes *is-mon*: *is-monoid M*
  shows *eval' M env* (*xs* @ *ys*) = *eval' M env xs* $\otimes_M$ *eval' M env ys*
  using *is-mon* **by** (*induction xs*, *auto simp*: *is-monoid-def*)

lemma *preserves-semantics*:
  assumes *is-mon*: *is-monoid M*
  shows *eval' M env* (*simpl e*) = *eval M env e*
  using *assms*
  unfolding *is-monoid-def*
proof (*induction e*)

case *One*
then show *?case*
by *simp*
next
case (*Var x*)
then show *?case*
by *auto*
next
case (*Plus e1 e2*)
then show *?case*
using *assms eval'-app*
by (*simp add*: *eval'-app*)
qed

Two expressions are equal in some monoid M, if they always evaluate to the same value for any environment:

definition *exps-equiv* :: *'a monoid* $\Rightarrow$ *Monoid* $\Rightarrow$ *Monoid* $\Rightarrow$ *bool*
(**infix** $\approx_1$ *60*) **where**
$e_1 \approx_M e_2 \equiv (\forall\ env.\ eval\ M\ env\ e_1 = eval\ M\ env\ e_2)$

The list monoid is just a record with append and the empty list for $\otimes$ and **1**:

definition *list-monoid* :: *'a list monoid* **where**
list-monoid $\equiv$ (| *mult* = (*op* @) , *one* = [] |)

It's a monoid:

lemma *list-monoid-is-monoid*:
is-monoid list-monoid
unfolding *is-monoid-def list-monoid-def*
by *auto*

We specialize the element type of the list to nat for later:

definition *list-monoid-nat* :: *nat list monoid* **where**
list-monoid-nat = *list-monoid*

We now define an environment that, if we evaluate a monoid expression in it, we don't lose any information. We know that we can build such an environment from the assumption that var is countably infinite

definition env_U :: *var* $\Rightarrow$ *nat list* **where**
$env_U\ x = [\mathcal{V}\ x]$

Since we don't want to manually unfold the definitions for the list monoid, we tell the simplifier to do this automatically.

declare *list-monoid-nat-def*[*simp*] *list-monoid-def*[*simp*]

We can show that evaluating the simplified expression is of the same length as the input list. This is fairly easy and could be solved by (induction xs, auto), but to show how this works, we can write out the proof in detail:

lemma *length-sim*:
  length (*eval′ list-monoid-nat* env_U *xs*) = *length xs*
proof (*induction xs*)
  case *Nil*

Here we have an empty list:

  hence *length* [] = *0* **by** *simp*

Simplifying the empty list gives us an empty list, by unfolding the definition of the list monoid.

  moreover have *eval′ list-monoid-nat* env_U [] = []
    by *simp*
  ultimately show *?case* **by** *simp*
next

In the other case we have that the list has one element x followed by list xs:

  case (*Cons x xs*)

We get from the induction hypothesis, we have that the length of xs is the same as when evaluating xs:

  then have *length xs* = *length* (*eval′ list-monoid-nat* env_U *xs*)
    by *simp*

Then adding an element in front of both will not change the length, because env_U by definition only returns singletons.

  then show *?case*
    unfolding env_U*-def*
    by *simp*
qed

Now we prove that we can "simulate" evaluating an expression without losing information:

lemma *eval-sim*:
  i < *length xs* $\Longrightarrow$ *eval′ list-monoid-nat* env_U *xs* ! i = $\mathcal{V}$ (*xs* ! i)
proof (*induction xs arbitrary*: *i*)

Again we use induction on xs.

  case *Nil*

There's no i such that i ¡ length [] = 0, so this is trivial:

  then show *?case*
    by *simp*
next
  case (*Cons x xs*)

This names the induction hypothesis 'Cons':

We can show this case by distinction on whether i is 0 or not:

then show *?case*
proof (*cases i*)
case *0*

This case follows trivially from the definition of $eval'$ and env_U

with env_U*-def* **show** *?thesis*
by *simp*
next
case (*Suc j*)

Here i = j + 1 for some j. Then the claim follows from the induction hypothesis, because it taking the ith element in $x \cdot xs$ will give the jth element in xs, since env_U returns single-element lists

with *Cons* **show** *?thesis*
unfolding env_U*-def*
by *simp*
qed
qed

Now we can proceed to the main proof: - Since Isabelle/HOL doesn't support quantifying over types, we specialize the equivalence assumption to the monoid for lists of natural numbers:

lemma *simpl-unique*:
assumes *eqv*: $e \approx_{\textit{list-monoid-nat}} e'$
shows $\textit{simpl } e = \textit{simpl } e'$
proof −

Some shorthands:

let $?e = \textit{simpl } e$
let $?e' = \textit{simpl } e'$
show *?thesis*
proof (*rule ccontr*)
assume *neq*: $\textit{simpl } e \neq \textit{simpl } e'$
have $\textit{length } ?e = \textit{length } ?e'$
proof (*rule ccontr*)
assume $\textit{length } ?e \neq \textit{length } ?e'$

Using $i < |xs| \Longrightarrow (eval' \textit{ list-monoid-nat } env_U \ xs)_{[i]} = \mathcal{V} \ xs_{[i]}$, this implies that the two unsimplified expressions also differ in length:

with *preserves-semantics* **have**
$\textit{length } (\textit{eval list-monoid-nat } env_U \ e) = \textit{length } (\textit{eval}' \textit{ list-monoid-nat } env_U \ (\textit{simpl } e))$
by (*metis list-monoid-is-monoid list-monoid-nat-def*)
moreover with *length-sim* **have**
$\textit{length } (\textit{eval list-monoid-nat } env_U \ e) \neq \textit{length } (\textit{eval list-monoid-nat } env_U \ e')$

by (*metis* ‹*length* (*simpl e*) $\neq$ *length* (*simpl e*′)› *list-monoid-is-monoid list-monoid-nat-def preserves-semantics*)
moreover from *preserves-semantics* **have**
length (*eval list-monoid-nat* env_U *e*′) = *length* (*eval*′ *list-monoid-nat* env_U (*simpl e*′))
by (*metis list-monoid-is-monoid list-monoid-nat-def*)
ultimately show *False*
using *length-sim*
by (*metis eqv exps-equiv-def*)
qed

Since the lengths are equal, we there must be at least one index i where they differ:

with *neq* **obtain** *i* **where** *in-list*: *i* < *length ?e* **and**
diff: *?e* ! *i* $\neq$ *?e*′ ! *i*
using *nth-equalityI* **by** *blast*
let *?x* = *?e* ! *i*
let *?y* = *?e*′ ! *i*

We have that looking up i in the unsimplified expression is the same as in the simplified one.

from *preserves-semantics* **have**
eval list-monoid-nat env_U *e* ! *i* =
eval′ *list-monoid-nat* env_U *?e* ! *i*
by (*metis list-monoid-is-monoid list-monoid-nat-def*)

The simplification lemma tells us that this is the same as applying $\mathcal{V}$ to the variable at that index:

moreover have *eval*′ *list-monoid-nat* env_U *?e* ! *i* = $\mathcal{V}$ *?x*
using ‹*i* < *length* (*simpl e*)› *eval-sim* **by** *blast*

By assumption this is different from the variable in e' at i:

moreover from *diff* **have** $\mathcal{V}$ (*?e* ! *i*) $\neq$ $\mathcal{V}$ (*?e*′ ! *i*)
using *var-countable* **by** *auto*

Analogously to before we know that this is the same as evaluating the unsimplified expressions.

moreover hence $\mathcal{V}$ (*?e*′ ! *i*) = *eval list-monoid-nat* env_U *e* ! *i*
by (*metis eqv eval-sim exps-equiv-def in-list length-sim list-monoid-is-monoid list-monoid-nat-def preserves-semantics*)

This means the evaluation results are different, at least at i, contradiction our equivalence assumption:

ultimately show *False*
by *linarith*
qed
qed

end