Supporting the Development of Dependently Typed Functional Programs

Sean Wilson
CISA, School of Informatics, The University of Edinburgh
sean.wilson@ed.ac.uk
Overview

• Programming with expressive dependent types can require the user and/or typechecker to construct proofs.

• We are developing generic proof automation to support programming with user defined inductive data types. We aim to support program properties relating to, for instance, non-linear arithmetic, permutations, membership and ordering.

• Using Coq/Russell as our programming environment, we are integrating a rippling tactic along with existing ones to automatically discharge proof obligations to make programming with dependent types easier.
Rippling

- As discharging Russell proof obligations can involve induction, rippling, a successful heuristic method for automating induction-like proofs, seems applicable.

- Basic idea: modify the conclusion to reduce its differences compared to a target term $T$, so $T$ can be used. $T$ is typically the inductive hypothesis in an inductive proof e.g.

  \[
  \text{Target: } \quad x + 1 = 1 + x
  \]

  \[
  \text{Conclusion: } \quad S(x + 1) = 1 + Sx
  \]

  Colour indicates differences

  \[
  \text{Differences are “rippled out”}
  \]

  \[
  S(1 + x) = 1 + Sx
  \]

  Fertilisation can then occur
Rippling/Generalisation

- Rippling guides induction variable/scheme choice, case splitting and lemma speculation.

- When fertilisation succeeds, generalising the goal and performing induction again (lemma calculation) is common.

- Generalisation is also helpful before we start induction on some proof obligations. This is common for proof obligations generated from the use of subset types.
Example: Rippling Applications

A list reversal function that admits it returns a permutation of its input:

Program Fixpoint count (a : list A) (x : A) : nat :=
match a with
  nil   => 0
| h :: t => if A_eq_dec h x then S (count t x) else (count t x)
end.

Program Fixpoint reverse (a : list A) :
  {o:list A | forall x, count a x = count o x} :=
match a with
  nil   => nil
| h::t => (reverse t)++(h::nil)
end.
Example: Rippling Applications

The recursive call of reverse generates this proof obligation:

\[
\begin{align*}
& h : A \\
& t : \text{list } A \\
& x : A \\
& x0 : \text{list } A \\
& e : \text{forall } x : A, \text{ count } t \ x = \text{ count } x0 \ x \\
& \text{--------------------------------------------} \\
& \text{count } (h :: t) \ x = \text{count } (x0 :: h :: \text{nil}) \ x
\end{align*}
\]

Rippling can be used to guide the proof as \( e \) is structurally similar to the conclusion.

After fertilisation and basic simplification, the following subgoals can be proven with induction and rippling:

\[
\begin{align*}
& S \ (\text{count } x0 \ x) = \text{count } (x0 :: x :: \text{nil}) \ x \\
& h <> x \rightarrow \text{ count } x0 \ x = \text{count } (x0 :: h :: \text{nil}) \ x
\end{align*}
\]
Example: Generalisation and Rippling Applications

A queue implementation with a function that admits items are queued in the intended order:

Definition queue := prod (list A) (list A).
Definition queue_to_list (q : queue) := (fst q) ++ (rev (snd q)).

Program Definition append_queue (a:queue) (b:queue):
{o:queue | queue_to_list o = queue_to_list a ++ queue_to_list b} :=
(fst a, (rev (queue_to_list b)) ++ (snd a)).

This generates the following proof obligation:
forall w x y z, x ++ rev (rev (w ++ rev y) ++ z) =
(x ++ rev z) ++ w ++ rev y

We can generalise the common subterm w ++ rev y to give:
forall x z g, x ++ rev (rev g ++ z) = (x ++ rev z) ++ g

We can then guide the inductive proof with rippling.
Developent Status

- A rippling out and common subterm generalisation tactic has been implemented.

- These have been integrated into a top-level tactic that recursively performs induction, rippling and generalisation.

- Can automatically produce proofs of many theorems involving Peano arithmetic and list functions e.g.

\[
\begin{align*}
\text{forall } l \ m \ n, \ (l \ ++ \ m) \ ++ \ n &= l \ ++ \ m \ ++ \ n. \\
\text{forall } x \ y, \ \text{rev} \ (x \ ++ \ y) &= \text{rev} \ y \ ++ \ \text{rev} \ x. \\
\text{forall } x, \ \text{rev} \ (\text{rev} \ x) &= x. \\
\text{forall } n \ m, \ m \ * \ n &= n \ * \ m. \\
\text{forall } m \ n \ k, \ (m \ + \ n) \ * \ k &= (m \ * \ k) \ + \ (n \ * \ k). \\
\text{forall } x \ n \ m, \ x^{(n+m)} &= x^n \ * \ x^m.
\end{align*}
\]
Future Work

- Continue rippling development: lemma speculation, allowing several target terms, exploiting universally quantified assumptions.

- Explore better generalisation algorithms and the use of counterexample checking.

- Evaluate the power of the proof automation.

- Explore how to provide useful feedback for programs that do not respect their typing.
Thanks for listening!