Proving Properties of Programs by Structural Induction by R. M. Burstall (*The Computer Journal*, 1969)

presented by
Jedidiah R. McClurg

Northwestern University

October 25, 2011

Introduction and Some Background

Why are we talking about proofs?

- The need for proofs of software correctness is becoming increasingly important.
  - Airline industry
  - Automotive industry
- Ultimately, we wish to automate the generation and/or checking of such proofs.
- Imperative languages such as C are especially difficult to reason about.

Simple C Code

```
int a = 0;
int b = a + 1;
```

Trickier C Code

```
int a = 0;
a = a + 1;
int b = a;
```

Functional Programming

How can we address these needs/difficulties?
Functional programming is one approach.

- This paper presents the language ISWIM, which is very similar to the modern language OCaml
- These have the following nice features:
  - Lambda calculus-style "let" bindings

    ```
    let a = 0 in
    let b = a + 1 in
    a + b;;
    ```
  - Algebraic data types (ADT)

    ```
    type tree = Tree of tree * int * tree | Leaf of int;;
    ```
  - Powerful expression-matching (important for the recursive paradigm)

    ```
    match t with
    | Leaf(i) -> print_string "found a leaf"
    | Tree(t1,i,t2) -> (* recursively process t1,t2 *)
    ```

Modeling Things Recursively

To take advantage of these nice features of functional programming, we must think recursively rather than iteratively!

- For example, consider the following simple algorithm to print each item in a list:

```
void print_list(int *l, int len) {
    for(int i = 0; i < len; i++) {
        printf("%d ", l[i]);
    }
}
```

- How would we do this recursively?

```
let rec print_list l =
    match l with
    | [] -> ()
    | a::ax ->
        print_int a; print_string " "; print_list ax
;;
```

Outline

The idea is that proofs regarding a recursively-structured program look very similar to the program itself (i.e. they are relatively straightforward to obtain)!

- First, we introduce some preliminaries used in the paper.
- Second, we present the idea of structural induction.
- Third, we show how to prove things via structural induction.
- Fourth (and finally), we examine some properties of an interesting sorting algorithm

Preliminaries

- We consider every expression in the (functional) program to be either an *atom* or a *structure* (i.e. an object built up from atoms).
- We can build structures using *construction operations*.
- Each construction operation has the following associated functions:
    - A constructor function to build up a new structure
    - A destructor function to get components of a structure
    - A predicate function to test for atomicity
- We define a *constituent* relation $<$ recursively as follows:

$$A < B \text{ iff } B = A \text{ or } A < b \text{ for some } b \in \mathit{components}(B)$$

(note that this a partial order).

A Basic Induction Principle

Here is the familiar induction principle:

Theorem (Induction)

Given a predicate $P(n)$ with $n \in \mathbb{N}$, if we have

1. $P(0)$ *is true*
2. $P(k) \implies P(k+1)$ *for arbitrary* $k \geqslant 0$,

then $P(n)$ is true for all $n \in \mathbb{N}$.

Proof.

This is a straightforward proof by contradiction (assume $P(j)$ is false for some $j > 0$ and see what happens). □

Stronger Induction Principle

Sometimes strengthening the induction hypothesis allows us to prove things more easily:

Theorem (Strong Induction)

Given a predicate $P(n)$ with $n \in \mathbb{N}$, if we have

1. $P(0)$ *is true*
2. $(\forall j < k, P(k)) \implies P(k+1)$ *for arbitrary* $k \geqslant 0$,

then $P(n)$ is true for all $n \in \mathbb{N}$.

Proof.

This is similar to the proof of the basic induction principle. $\square$

Structural Induction Principle

Induction is not limited to predicates of natural numbers. Consider the Structural Induction principle, as put forth in Burstall's paper:

Theorem (Structural Induction)

Given a set S of structures and a property $P(s)$ for $s \in S$, if we have

$$(\forall c \in constituents(s), P(c)) \implies P(s) \textit{ for arbitrary } s \in S,$$

then $P(s)$ is true for all $s \in S$. (Note the "hidden" base case!)

Proof.

This proof follows the same line of reasoning as the other induction principles. Structures are built up using finitely many construction operations. $\square$

Simple Proofs Using Structural Induction

Now, we are ready to begin proving things about recursive programs. Let's consider the following LISP-like constructs:

- nil: a null atom
- cons: concatenate (i.e. join together a car and cdr)
- car: get first item (i.e. destruct a cons)
- cdr: get remainder of cons

We can do list operations with these, e.g.

```
cons(a,cons(b,cons(c,nil)))
car(cons(d,nil))
```

Simple Proofs Using Structural Induction (Cont.)

Calling *cons* and *nil* by their more common names **::** and **[]**, we can define some useful recursive functions:

```
let rec concat xs1 xs2 =
match xs1 with
| [] -> xs2
| x::xs -> x::(concat xs xs2) ;;

let rec lit f xs1 y =
match xs1 with
| [] -> y
| x::xs -> f x (lit f xs y) ;;
```

Note that the second function is similar to OCaml's *fold* function(s).

Simple Proofs Using Structural Induction (Cont.)

Let's prove something about these functions.

Theorem (Fold and Concat)

(lit f (concat xs1 xs2) y) = (lit f xs1 (lit f xs2 y))

Proof.

We begin the proof with induction on the structure of xs1. Since there is only one atom (nil) and one constructor (cons), we have two choices for the structure of xs1

1. xs1 is of the form nil
   - We can simply expand the definitions to get (lit f xs2 y) = (lit f xs2 y)
2. xs1 is of the form x::xs
   - Here our induction hypothesis states that the theorem holds for xs. We proceed as follows...

□

Simple Proofs Using Structural Induction (Cont.)

Theorem (Fold and Concat, Continued)

(lit f (concat xs1 xs2) y) = (lit f xs1 (lit f xs2 y))

Proof.

- We can transform the LHS of the theorem into (lit f (x :: (concat xs xs2)) y) by the definition of concat
- We can further transform this into (f x (lit f (concat xs xs2) y)) by the definition of lit.
- Now, we can transform the RHS of the theorem into (f x (lit f xs (lit f xs2, y))) by the definition of lit.
- We can further transform this into (f x (lit f (concat xs xs2) y)) by applying our inductive hypothesis in regards to xs.
- Thus, LHS = RHS.

□

More Interesting Proofs

Consider the following implementation of Merge Sort:

```
let rec merge al bl = match (al,bl) with
    | ([],_) -> bl  | (_,[]) -> al
    | (a::ax,b::bx) ->
        (if (a < b) then a::(merge ax bl) else
                         b::(merge al bx)) ;;

let rec mergesort l = match l with
    | [] -> []  | a::[] -> l
    | a ->
       let (left, right) = split a in
       let ls = mergesort left in
       let rs = mergesort right in
       merge ls rs ;;
```

Prove that `merge` returns a sorted list when given two sorted lists.

More Interesting Proofs (Cont.)

The paper goes on to prove the correctness of a tree sorting algorithm, and a small compiler for a simple stack-based machine. All of these proofs adhere to the following paradigm:

1. Represent your data and operations as algebraic data types and recursive constructor functions.
2. To prove a property about all data, prove the property for atomic data and then prove the property under the assumption that it holds for subdata.

Conclusion

- Structural induction is a useful method of proving things about recursive programs.
- Functional programming is a usable and natural way to define and reason about programs via structural induction.

Thanks!