Lecture 29: Difficult proofs
Definitions & match statements

Sebastiaan Joosten

University of Minnesota
Driven to Discover®

Outline

- Why reasoning about expressions with ‘match’ is hard
- Avoiding rewrite-loops…
  - … and why reasoning with definitions is hard
- How to deal with both

Why reasoning with matches is hard…

- match (Cons (foo x y, z)) with
  | Cons (Cons (h,tl), t2) -> bar
  | Cons (Nil, tl) -> baz
  | Cons (foo x y, t3) -> bozo
  | x -> foo
  = ??
- Can we rewrite the above match statement because it matches a pattern?

Why reasoning with matches is hard…

- match (Cons (foo x y, z)) with
  | Cons (Cons (h,tl), t2) -> bar
  | Cons (Nil, tl) -> baz
  | Cons (foo x y, t3) -> bozo
  | x -> foo
  = ??
- A: we cannot make a step: both ‘bar’ and ‘baz’ could be returned, depending on what ‘foo x y’ returns.

Orthogonal patterns

- Call a list of match statements ‘orthogonal’ if at most one of them can apply:
- Examples of orthogonal left hand sides:
  - | Cons (h,tl) -> …
    | Nil -> …
  - | Leaf -> …
    | Node (Leaf, el, Leaf) -> …
    | Node (Node (x,y,z), el, Node(x,y,z) -> …
- Examples of non-orthogonal left hand sides:
  - | Cons (h, tl) -> …
    | _ -> …
  - | Node (Leaf, el _) -> …
    | Node (_, el, Leaf) -> …

Orthogonal patterns

- For (closed) finite types, a non-orthogonal pattern-list can always be turned into one that is orthogonal:
  - use the ‘earliest match first’ rule
  - might turn as few as 2 patterns into *many* (no upper bound)
  - we’ll not implement this algorithm (your users will need to do it by hand)

Why reasoning with matches is hard (2)

- match (Cons (x, y)) with
  | Cons (y, z) -> foo x y
  = ??

Why reasoning with matches is hard (2)

- match (Cons (x, y)) with
  | Cons (y, z) -> foo x y
  =
  foo x x

- This is called variable renaming
  it's not a problem per-se

Why reasoning with matches is hard (3)

- Another variable naming problem, consider this rule:
- foo x y = match x with
  | Cons (a, b) -> foo y b
  | Nil -> x
- We cannot always apply this rule naively:
- foo (Cons (1, 2)) a
  =
  match (Cons (1,2)) with
  | Cons (a, b) -> foo a b
  | Nil -> x
- this is called ‘variable capture’, and it’s incorrect!
- … Solution: rename variables within the match

Why match is hard (summary)

- patterns might be overlapping
- variables may need to get renamed
- variables should not get ‘captured’

Dealing with…

Infinite rewrites

Infinite rewrites: the problem

- Consider this rule about ‘canonical form’:
- cf x = cf (cf x)

- Suppose we want to prove that ‘cf (cf x) = cf (cf (cf x))’:
- starting on the left:
  cf (cf x)
  =
  cf (cf (cf x))
  =
  cf (cf (cf (cf x)))
  =
  … (and so on)

Ideas on fixing this?

Infinite rewrites: the problem

- Consider this rule about ‘canonical form’:
- cf (cf x) = cf x

- Suppose we want to prove that ‘cf (cf x) = cf (cf (cf x))’:
- starting on the left:
  cf (cf x)
  =
  cf x

- starting on the right:
  cf (cf (cf x))
  =
  cf (cf x)
  =
  cf x

What’s the final proof?

Infinite rewrites: the solution

- For any rule lhs = rhs
  ensure that rhs is simpler than the lhs
- For example:
  - cf (cf x) = cf x is good
  - cf x = cf (cf x) is bad
- We can re-orient these rule by hand
- We'll ignore cases where this is not possible:
  - foo x y = foo y x is bad
  - foo y x = foo x y is bad (exact same thing)

This nearly solves the infinite rewrites, if it wasn't for …

… definitions

Definitions: the problem

- A definition *typically* has a more complicated rhs:

```
let rec append (l1 : list) (l2 : list) : list =
  match l1 with
   | Nil -> l2
   | Cons ((h : int), (t : list))
      -> Cons (h, append t l2)
```

- Corresponding rule:

```
append l1 l2
  = match l1 with
   | Nil -> l2
   | Cons ((h : int), (t : list))
      -> Cons (h, append t l2)
```

Definitions make bad rules

- We get problems no matter how we orient the definition of append:

```
append l1 l2
  = match l1 with
  | Nil -> l2
  | Cons ((h : int), (t : list))
     -> Cons (h, append t l2)

or:

match l1 with
  | Nil -> l2
  | Cons ((h : int), (t : list))
     -> Cons (h, append t l2)
  = append l1 l2
```

Definitions make bad rules

- regular direction:

```
append l1 l2
  =
match l1 with
  | Nil -> l2
  | Cons ((h : int), (t : list))
     -> Cons (h, append t l2)
  =
match l1 with
  | Nil -> l2
  | Cons ((h : int), (t : list))
     -> Cons (h, (match t with
  | Nil -> l2
  | Cons ((h : int), (t : list))
     -> Cons (h, append t l2)) t l2)
  = …
```

Definitions make bad rules

- reverse direction:

```
match l1 with
   | Nil -> l2
   | Cons ((h : int), (t : list))
      -> Cons (h, append t l2)
  = append l1 l2
```

- Either never applies because there's no sensible way to get a 'match'
- Or it applies on everything because it matches the 'l2' case:
  l2 = append Nil l2 = append Nil (append Nil l2) = …

Definitions: two solutions

- Both of these would solve most issues:
  - Only apply a definition only if no non-definitions apply, and at most once
  - Instead of applying the definition proper, unravel the match rule, and apply the definition together with the ‘match’ rule

- The first is easier to implement by itself, but leaves us with a ‘match’ after applying a definition
- The last solution enables match-free reasoning, and hence easier to implement overall

Solving both problems with one solution

A typical definition

- let rec foo (x : ..) (y : ..) = match x with
  | Nil -> … some expression (1) …
  | Cons (h,tl) -> … some expression (2) …
- Since the patterns are *orthogonal*, we can turn this into two rules:
- foo Nil y = … some expression (1) …
  foo (Cons (h,tl)) y = … some expression (2) …
- Notes:
  - Each of the resulting rules is a definition + match step.
  - Our ‘x’ has gone missing, it might occur in (1) or (2).

Implementing definitions

- Suppose a definition is of this shape:
  - let rec *name arguments* = match *arg* with
    | … list of patterns and expressions …
- Moreover, suppose the list of patterns is *orthogonal*.
- Then for each pattern:
  pat -> *exp*
  - we can instead introduce the rule: *name arguments* = *exp*
  - where **both sides** of this rules have all occurrences of *arg* replaced by *pat*.
- Why replace both sides?
- What are examples of definitions not covered?

Implementing definitions

- Suppose a definition is of this shape:
  - let rec *name arguments* = match *arg* with
    | … list of patterns and expressions …
- Moreover, suppose the list of patterns is *orthogonal*.
- Then for each pattern:
  pat -> exp
  - we can instead introduce the rule: *name arguments* = *exp*
  - where **both sides** of this rules have all occurrences of *arg* replaced by *pat*.
- Why replace both sides?
  A: after replacing *arg*, it doesn't occur on the left, but it might on the right
- What are examples of definitions not covered?
  A: match not at top level / match e with where e is not an argument

Implementing definitions: extra fancy ideas (not required)

- Apply our idea more generally for all rules of the form:
  - *expr* = match *variable* with …
- This allows us to deal with nested match statements!

- match doesn't have to be at top level!
  - Suppose a rule is of this shape:
    foo x y = fn (match x with … | pattern -> expression)
  - We can create this rule instead:
    foo pattern y = fn expression
    (again: remember to replace x with pattern on *both sides*)