

Recursion Pattern Analysis and Feedback

Sander Mak Thomas van Noort

Center for Software Technology, Universiteit Utrecht

October 27, 2006



Overview

- 1 Introduction
- 2 Recognizing Patterns
- 3 Feedback
- 4 Other applications
- 5 Implementation
- 6 Related Work
- 7 Conclusion



Why focus on recursion?

Recursion is an essential part of Haskell:

it is *the only way to loop or iterate*



Why focus on recursion?

Recursion is an essential part of Haskell:

it is *the only way to loop or iterate*

Sometimes this message is taken to heart. Or too hard...

Naive recursion

$$f [] = 1$$

$$f (x : xs) = (x + 1) * f xs$$



Why focus on recursion?

Recursion is an essential part of Haskell:

it is *the only way to loop or iterate*

Sometimes this message is taken to heart. Or too hard...

Naive recursion

$$f [] = 1$$

$$f (x : xs) = (x + 1) * f xs$$

Pretty recursion

$$f = foldr (*) 1 . map (+1)$$

or :

$$f = foldr (\lambda x xs \rightarrow (x + 1) * xs) 1$$


Our goal

Eliminate Haskell-recursion illiteracy:

IRC transcript

```

<CosmicRay> so let's say I'm writing a recursive function like this:
<CosmicRay> myFoo [] = []
<CosmicRay> myFoo (x:xs) = (x + 5) : myFoo xs
<JaffaCake> i.e. map (+5)
<CosmicRay> right, but I don't know how map works internally,
so I wrote it out
<JaffaCake> it works just like that ;)
  
```

(and improve life for Bastiaan by doing so)



Research questions

Improvement of recursive definitions is quite a broad goal. More concrete sub-questions are:

- Which recursion combinators are available?
 - Usual suspects: *foldr*, *map*
 - But also: *filter*, *foldl*, etc.
- Which patterns do we want to recognize?
 - Are some functions better of in a naive definition?
 - Where are recursion patterns described?
- How do we present these improvements?
- Where do we present these improvements?



What constitutes a pattern?

No formal definition (yet), but there are several different sides:

Syntactic pattern:

$$f \quad \quad \quad :: [Int] \rightarrow Int$$
$$f [] \quad \quad = 0$$
$$f (x : xs) = x + f xs$$


What constitutes a pattern?

No formal definition (yet), but there are several different sides:

Syntactic pattern:

$$f \quad \quad \quad :: [Int] \rightarrow Int$$

$$f [] \quad \quad = 0$$

$$f (x : xs) = x + f xs$$

$$f = foldr (\lambda x xs \rightarrow x + xs) 0$$


What constitutes a pattern?

No formal definition (yet), but there are several different sides:

Syntactic pattern:

$$f \quad \quad \quad :: [Int] \rightarrow Int$$

$$f [] \quad \quad = 0$$

$$f (x : xs) = x + f xs$$

$$f = foldr (\lambda x xs \rightarrow x + xs) 0$$

Pure syntactic matching will not do, there need to be semantic restrictions:

- No recursion in base case (and even: what is a base case/recursive case?)
- Example: x should not be involved in recursive call
- What if more parameters are present in definition?



Generalizing patterns

Previous pattern was specialized for lists.

Alternative list

```
data FastList a = Nil
    | Cons    a (FastList a)
    | Unit    a
    | Append (FastList a) (FastList a)
```

Ideally, we want to recognize recursive definitions over arbitrary datatypes.

- Is there a general pattern? (e.g. for *folds*?)
- Even if so, do we want to give feedback? (There might be no *fold* for this datatype)
- Are there usecases for user-defined patterns?
- What about overlapping patterns in recursive definition?



Constructing feedback

Form of the feedback seems to be straightforward:

- Report improved definition of a naive recursive function



Constructing feedback

Form of the feedback seems to be straightforward:

- Report improved definition of a naive recursive function

Why?

- 1 Learn new constructs to user
- 2 Enhance sense of beauty
- 3 Using recursion primitives enhances maintainability
- 4 Efficiency? (*foldl'* vs. *foldr*)



Constructing feedback

Form of the feedback seems to be straightforward:

- Report improved definition of a naive recursive function

Why?

- 1 Learn new constructs to user
- 2 Enhance sense of beauty
- 3 Using recursion primitives enhances maintainability
- 4 Efficiency? (*foldl'* vs. *foldr*)

Where and how:

- At compile time as *'warning'*
- Alternatively: indicate in development environment (like Eclipse FP)
- Point to source location, or include original expression in feedback (or both)



Applications besides educational

Are there other useful applications?



Applications besides educational

Are there other useful applications?

optimization

Knowledge whether function is a *fold* (catamorphic) can be useful:

- To decide on inlining code (termination)
- Deforestation (Warm Fusion)



The *Fun*-language

Small functional language based on expressions:

Concrete syntax of expressions

$e ::= n$	constants
v	variables
$e_1 \oplus e_2$	infix operators
$\lambda p_1 \dots p_n \rightarrow e$	lambda abstraction
fix $v e$	function definition
$e_1 e_2$	application
case v of $p_1 \rightarrow e_1; \dots p_n \rightarrow e_n$	case analysis
$[]$	empty list
$e_1 : e_2$	list constructor



The *Fun*-language

Small functional language based on expressions:

Concrete syntax of expressions

$e ::= n$	constants
v	variables
$e_1 \oplus e_2$	infix operators
$\lambda p_1 \dots p_n \rightarrow e$	lambda abstraction
fix $v e$	function definition
$e_1 e_2$	application

Concrete syntax of patterns

$p ::= v$	binding
$p_1 : p_2$	(nested) list
$[]$	empty list
$-$	wild card



Example recursive definition

Haskell definition of *sum*

```

sum      :: [Int] → Int
sum []   = 0
sum (x : xs) = x + sum xs

```

Fun definition of *sum*

```

fix sum λl → case l of [] → 0;
                x : xs → x + sum xs

```

- **fix** construct for recursive function definitions
- No pattern matching with multiple bodies in abstraction possible, an additional case expression is required



Main functionality

Two main functions for analyzing an expression:

- $suggest :: String \rightarrow IO ()$
- $analyze :: Expr \rightarrow Maybe Expr$

A number of assumptions have been made:

- A **fix**\ λ **case** construct appears at top-level
- Expressions to analyze are type correct
- Expressions do not contain shadowed identifiers
- Only (some) *foldr* patterns on lists are recognized...
... and some experimental *map* patterns

We will reflect on these assumptions later on.



Completeness and soundness

We do not claim to be complete, nor is it our goal:

- Not recognizing a construct will never result in errors
- Not all suggestions are necessarily better
- What is the benchmark of completeness anyway?
- Each new construct adds complexity

We do however want to be sound, every suggestion given should be

- Type correct
- Of the same type as the original expression
- Equal to original expression (extensional equality)

And, everything we recognize is a valid recursion pattern



Analyzing *length*

*length*_{Expr}: the length of a list

```
fix length  $\lambda l \rightarrow$  case l of [ ]  $\rightarrow 0$ ;  
                  x : xs  $\rightarrow 1 +$  length xs
```



Analyzing *length*

$length_{Expr}$: the length of a list

```
fix length  $\lambda l \rightarrow$  case  $l$  of  $[] \rightarrow 0;$   

 $x : xs \rightarrow 1 + length\ xs$ 
```

Suggested definition for *length*

*FoldrRecognizer> *suggest* $length_{Expr}$

A more elegant solution would be:

```
foldr ( $\lambda\_ xs \rightarrow 1 + xs$ ) 0
```



Analyzing *sum*

sum_{Expr} : sum elements of list

```
fix sum  $\lambda l \rightarrow$  case l of [ ]  $\rightarrow 0$ ;  
                  x : xs  $\rightarrow x + sum$  xs
```



Analyzing *sum*

sum_{Expr} : sum elements of list

```
fix sum  $\lambda l \rightarrow$  case l of [ ]  $\rightarrow 0$ ;
                x : xs  $\rightarrow x + sum$  xs
```

Suggested definition for *sum*

*FoldrRecognizer> *suggest* sum_{Expr}

A more elegant solution would be:

foldr (+) 0



Analyzing *map*

*map*_{Expr}: mapping a function over a list

```
fix map  $\lambda f l \rightarrow$  case l of []  $\rightarrow$  [];  
                x : xs  $\rightarrow$  f x : map f xs
```



Analyzing *map*

*map*_{Expr}: mapping a function over a list

```
fix map λf l → case l of [ ] → [ ];
                x : xs → f x : map f xs
```

Suggested definition for *map*

```
*FoldrRecognizer> suggest mapExpr
```

A more elegant solution would be:

```
λf → foldr (λx xs → f x : xs) [ ]
```

- Note that the *definition* of *map* is recognized, not the *usage*



Analyzing *append*

*append*_{Expr}: appending two lists (++)

```
fix append  $\lambda l_1 l_2 \rightarrow$  case  $l_1$  of [ ]  $\rightarrow l_2$ ;
                                $x : xs \rightarrow x : \textit{append} \textit{xs} l_2$ 
```

Suggested definition for *append*

*FoldrRecognizer> *suggest append*_{Expr}

A more elegant solution would be:

```
foldr ( $\lambda x \textit{xs} \rightarrow \lambda l_2 \rightarrow x : \textit{xs} l_2$ ) ( $\lambda l_2 \rightarrow l_2$ )
```



Main algorithm

The main algorithm consists of the following steps:

- Pre-process function definitions with multiple arguments
- Analyze that the definition can be written using *foldr*
- If so,
 - Construct the new definition using *foldr*
 - Post-process the returned lambda abstractions
 - Generate feedback
- Otherwise,
 - Generate feedback



Pre-processing

The analysis is based on a function with a single argument.

Original definition of *map*

```
fix map  $\lambda f l \rightarrow$  case l of []  $\rightarrow$  [];  
                  x : xs  $\rightarrow$  f x : map f xs
```



Pre-processing

The analysis is based on a function with a single argument.

Original definition of *map*

```
fix map λf l → case l of [ ] → [ ];
                x : xs → f x : map f xs
```

Pre-processed definition of *map*

```
λf → fix map λl → case l of [ ] → [ ];
                x : xs → f x : map xs
```

- Arguments *before* the list argument are moved to the front



Pre-processing (2)

The analysis is based on a function with a single argument.

Original definition of *append*

```
fix append  $\lambda l_1 l_2 \rightarrow$  case  $l_1$  of [ ]  $\rightarrow l_2;$   

 $x : xs \rightarrow x : \text{append } xs l_2$ 
```



Pre-processing (2)

The analysis is based on a function with a single argument.

Original definition of *append*

```
fix append  $\lambda l_1 l_2 \rightarrow$  case  $l_1$  of [ ]  $\rightarrow l_2;$   

 $x : xs \rightarrow x : \textit{append} \ x \ l_2$ 
```

Pre-processed definition of *append*

```
fix append  $\lambda l_1 \rightarrow$  case  $l_1$  of [ ]  $\rightarrow \lambda l_2 \rightarrow l_2;$   

 $x : xs \rightarrow \lambda l_2 \rightarrow x : \textit{append} \ x \ l_2$ 
```

- Arguments *after* the list argument are added to each case



Pre-processing (3)

Moving all arguments to the front is semantically incorrect:

Example definition

```
fix weird  $\lambda l_1 l_2 l_3 \rightarrow$  case  $l_2$  of [ ]  $\rightarrow l_1;$   
 $x : xs \rightarrow$  append  $l_3$  (weird  $l_1$   $xs$   $l_3$ )
```



Pre-processing (3)

Moving all arguments to the front is semantically incorrect:

Example definition

$$\mathbf{fix} \textit{ weird} \lambda l_1 l_2 l_3 \rightarrow \mathbf{case} l_2 \mathbf{of} [] \rightarrow l_1;$$

$$x : xs \rightarrow \mathit{append} l_3 (\textit{weird} l_1 xs l_3)$$

Incorrect pre-processed definition of *weird*

$$\lambda l_1 l_3 \rightarrow \mathbf{fix} \textit{ weird} \lambda l_2 \rightarrow \mathbf{case} l_2 \mathbf{of} [] \rightarrow l_1;$$

$$x : xs \rightarrow \mathit{append} l_3 (\textit{weird} xs)$$

- Order of arguments must remain unchanged!
- Rest of the algorithm is applied to the expression without the abstraction in front (it is added in the end)



Pre-processing (3)

Moving all arguments to the front is semantically incorrect:

Example definition

$$\mathbf{fix\ weird}\ \lambda l_1\ l_2\ l_3 \rightarrow \mathbf{case}\ l_2\ \mathbf{of}\ [\] \quad \rightarrow l_1;$$

$$x : xs \rightarrow \mathit{append}\ l_3\ (\mathit{weird}\ l_1\ xs\ l_3)$$

Correct pre-processed definition of *weird*

$$\lambda l_1 \rightarrow \mathbf{fix\ weird}\ \lambda l_2 \rightarrow \mathbf{case}\ l_2\ \mathbf{of}\ [\] \quad \rightarrow \lambda l_3 \rightarrow l_1;$$

$$x : xs \rightarrow \lambda l_3 \rightarrow \mathit{append}\ l_3\ (\mathit{weird}\ xs\ l_3)$$

- Order of arguments must remain unchanged!
- Rest of the algorithm is applied to the expression without the abstraction in front (it is added in the end)



Analysis

Expressions can be defined using *foldr* iff:

- A **fix**\λ\ **case** construct appears at top-level
- The **case** construct has exactly two cases
 - A base pattern (`[]`)
 - A recursive pattern (`x : xs`)
- The base case does not use recursion
- The recursive case does use recursion
- The recursive case does not use the complete list argument

$$\mathbf{fix\ tails\ } \lambda l \rightarrow \mathbf{case\ } l \mathbf{ of\ } [] \rightarrow [] : [] \\ x : xs \rightarrow l : \mathbf{tails\ } xs$$

$$\mathbf{foldr\ } (\lambda x\ xs \rightarrow l : xs) ([] : [])$$


Analysis (2)

Properties defined compositionally for reuse:

Recognizing a *foldr*

isFoldr :: *Expr* → *Bool*

isFoldr *expr* = *and* (*map* (\$) *expr*) *props*)

where *props* = [*isFunDef*
 , *hasNrOfCases* 2
 , *hasPattern* *isNilPat*
 , *hasPattern* *isConsPat*
 , *not* . *caseUses* *getBaseCase* *getFunIdent*
 , *caseUses* *getRecCase* *getFunIdent*
 , *not* . *caseUses* *getRecCase* *getCaseArg*
]



Constructing a new definition

Body of cases (almost) directly map to arguments of *foldr*:

- Body of base case is 2nd argument of *foldr*
- Body of recursive case is 1st argument of *foldr*, but requires:
 - Replacing recursive call with the tail of the original argument list

Pre-processed definition of *append*

$$\mathbf{fix} \text{ append } \lambda l_1 \rightarrow \mathbf{case} \ l_1 \ \mathbf{of} \ [] \quad \rightarrow \lambda l_2 \rightarrow l_2;$$

$$x : xs \rightarrow \lambda l_2 \rightarrow x : \text{append } xs \ l_2$$


Constructing a new definition

Body of cases (almost) directly map to arguments of *foldr*:

- Body of base case is 2nd argument of *foldr*
- Body of recursive case is 1st argument of *foldr*, but requires:
 - Replacing recursive call with the tail of the original argument list

Pre-processed definition of *append*

```
fix append  $\lambda l_1 \rightarrow$  case  $l_1$  of [ ]  $\rightarrow \lambda l_2 \rightarrow l_2;$   

 $x : xs \rightarrow \lambda l_2 \rightarrow x : \text{append } xs\ l_2$ 
```



Constructing a new definition

Body of cases (almost) directly map to arguments of *foldr*:

- Body of base case is 2nd argument of *foldr*
- Body of recursive case is 1st argument of *foldr*, but requires:
 - Replacing recursive call with the tail of the original argument list

Pre-processed definition of *append*

$$\mathbf{fix} \text{ append } \lambda l_1 \rightarrow \mathbf{case} \ l_1 \ \mathbf{of} \ [] \quad \rightarrow \lambda l_2 \rightarrow l_2;$$

$$x : xs \rightarrow \lambda l_2 \rightarrow x : \text{append } xs \ l_2$$


Constructing a new definition

Body of cases (almost) directly map to arguments of *foldr*:

- Body of base case is 2nd argument of *foldr*
- Body of recursive case is 1st argument of *foldr*, but requires:
 - Replacing recursive call with the tail of the original argument list

Pre-processed definition of *append*

```
fix append  $\lambda l_1 \rightarrow$  case  $l_1$  of [ ]  $\rightarrow \lambda l_2 \rightarrow l_2;$   

 $x : xs \rightarrow \lambda l_2 \rightarrow x : \text{append } xs l_2$ 
```



Constructing a new definition

Body of cases (almost) directly map to arguments of *foldr*:

- Body of base case is 2nd argument of *foldr*
- Body of recursive case is 1st argument of *foldr*, but requires:
 - Replacing recursive call with the tail of the original argument list

Pre-processed definition of *append*

$$\mathbf{fix} \text{ append } \lambda l_1 \rightarrow \mathbf{case} \ l_1 \ \mathbf{of} \ [\] \quad \rightarrow \lambda l_2 \rightarrow l_2;$$

$$x : xs \rightarrow \lambda l_2 \rightarrow x : \text{append } xs \ l_2$$

New definition of *append*

$$\text{foldr} \ (\lambda x \ xs \rightarrow \lambda l_2 \rightarrow x : xs \ l_2) \ (\lambda l_2 \rightarrow l_2)$$


Post-processing

Derived arguments of a call to *foldr* can be simplified:

- Infix operators

Suggested definition for *sum*

```
foldr ( $\lambda x\ xs \rightarrow x + xs$ ) 0
```

Simplified definition of *sum*

```
foldr (+) 0
```



Post-processing

Derived arguments of a call to *foldr* can be simplified:

- Infix operators

Suggested definition for *sum*

```
foldr ( $\lambda x\ xs \rightarrow x + xs$ ) 0
```

Simplified definition of *sum*

```
foldr (+) 0
```

- Function applications

Suggested definition for *concat*

```
foldr ( $\lambda x\ xs \rightarrow \text{append } x\ xs$ ) [ ]
```

Simplified definition of *concat*

```
foldr append [ ]
```



Post-processing (2)

- Collapsing lambda abstractions (not implemented yet)

Suggested definition for *append*

$$\text{foldr } (\lambda x \text{ xs } \rightarrow \lambda l_2 \rightarrow x : \text{xs } l_2) (\lambda l_2 \rightarrow l_2)$$

Simplified definition of *append*

$$\text{foldr } (\lambda x \text{ xs } l_2 \rightarrow x : \text{xs } l_2) (\lambda l_2 \rightarrow l_2)$$


Post-processing (2)

- Collapsing lambda abstractions (not implemented yet)

Suggested definition for *append*

$$\text{foldr } (\lambda x \text{ xs } \rightarrow \lambda l_2 \rightarrow x : \text{xs } l_2) (\lambda l_2 \rightarrow l_2)$$

Simplified definition of *append*

$$\text{foldr } (\lambda x \text{ xs } l_2 \rightarrow x : \text{xs } l_2) (\lambda l_2 \rightarrow l_2)$$

- Replacing unused bindings with wild card

Suggested definition for *length*

$$\text{foldr } (\lambda x \text{ xs } \rightarrow 1 + \text{xs}) 0$$

Simplified definition of *length*

$$\text{foldr } (\lambda _ \text{ xs } \rightarrow 1 + \text{xs}) 0$$


Dr. Haskell

A tool to detect common patterns in Haskell code:

Example definition

```
f n x = take n (repeat x)
```

Suggestion

```
f n x = replicate n x
```

Implemented by normalizing code and matching hard-coded patterns:

Defined pattern

```
use_replicate n x = take n (repeat x)
```



Dr. Haskell (2)

Developed by *Neil Mitchell* at the *University of York*

- Implemented using *Yhc*
- Only recognizes patterns in function application...
... and lacks recognizing patterns in function definitions
- Only based on syntactic restrictions...
... and lacks semantic restrictions
- Only points to source location which matches a pattern...
... instead of giving a suggestion

Not really suited for recognizing recursion patterns in general



General observations

The current implementation is restricted to:

- Expressions without shadowed identifiers
- Recognizing (some) *foldr* patterns on lists, reusability of certain components has yet to be proven
- Expressions with a **fix**\λ\case construct at top-level
- Simplifying just some lambda abstractions
- Always just returning a more 'elegant' solution, although this is not always the case

Suggested definition for *weird*

$$\lambda l_1 \rightarrow \text{foldr } (\lambda _ xs \rightarrow \lambda l_3 \rightarrow \text{append } l_3 (xs \ l_3)) (\lambda l_3 \rightarrow l_1)$$

The implementation is not complete at all, but believed to be sound...



Future work

First thing to do, try to recognize other recursion patterns:

- Is it easy to implement recognizing other recursion patterns?
- What parts can be reused?
- Is there a general structure which can be abstracted over?

Keeping quality of feedback in mind of course...

- Which suggestions are shown?
- Reducing the complexity of the suggestions by simplifying



Concluding remarks

- No supporting theory available, so our approach consists of
 - Trial-and-error
 - Use of unit tests
 - Lots of hard thinking...
- Translation to mature language
 - Requires normalisation of expressions to be analyzed
 - Normalisation should not interfere with feedback
 - Giving feedback requires pointing to source location
- Lambda simplification improves the quality of feedback
- Current feedback would already be useful for 1st-year FP students



Discussion

- Should prelude functions (e.g. *const*, *flip* and *id*) be used in our feedback?



Discussion

- Should prelude functions (e.g. *const*, *flip* and *id*) be used in our feedback?
- What determines the complexity of a suggestion?



Discussion

- Should prelude functions (e.g. *const*, *flip* and *id*) be used in our feedback?
- What determines the complexity of a suggestion?
- Would you expand detection to arbitrary datatypes (width) or expand by adding new patterns over the *List*-datatype (depth)?



Discussion

- Should prelude functions (e.g. *const*, *flip* and *id*) be used in our feedback?
- What determines the complexity of a suggestion?
- Would you expand detection to arbitrary datatypes (width) or expand by adding new patterns over the *List*-datatype (depth)?
- Which patterns would be useful to detect?

