

# Proposal – A Refactoring to Add a Constructor to a Data Type

Christopher M Brown

February 10, 2006

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
<b>2</b>	<b>Adding a Constructor to a Data Type</b>	<b>6</b>

# List of Figures

2.1	An example function for program slicing . . . . .	7
2.2	An application of program applied to the sub-expression res . . . . .	7
2.3	There is already a catch all in the pattern match . . . . .	7
2.4	There is already a catch all in the pattern match - added a new clause . . . . .	7
2.5	An example showing a function created from the data dependencies . . . . .	8
2.6	A simple data type . . . . .	8
2.7	data type with a constructor added . . . . .	9
2.8	A function which takes two data types as parameters . . . . .	9
2.9	A function which takes two data types as parameters . . . . .	9
2.10	A function which takes two data types as parameters . . . . .	9
2.11	The data type Sign . . . . .	10
2.12	A function game which works with two Sign parameters. . . . .	10
2.13	The data type Sign . . . . .	10
14	Shows a simple case where Newcastle has been added to Weather . . . . .	13
15	adding the constructor Consett to Weather in this example should result in programatica giving an error . . . . .	14
16	This file shows how getTemp works over the data type Weather the next figure shows a constructor being added to Weather . . . . .	14
17	Consett has been added to Weather and the refactoring has added a wildcard to the function getTemp . . . . .	15
18	The refactoring also needs to take into account class instances, and pattern matches that occur within equations in the class instances . . . . .	16
19	The refactoring also needs to take into account class instances. The constructor Raining has been added to Temp and an undefined clause has been added to the Print instance for Temp . . . . .	17
20	This example shows a case where the data type uses field names. In this case the refactorer should use field names if and only if they have been declared in the data type with field names. . . . .	18
21	This example shows where a case pattern matching is used with multiple arguments and only one of the arguments are of the type in question. The pattern matching matches every constructor from one data type with every constructor from the data type in question . . . . .	20
22	This example extends the case from the previous figure. A new constructor has been added to the data type <code>Weather</code> , the refactoring keeps the consistencies seen in the other function pattern matches but ignores the pattern matching in the case expression . . . . .	21

23	This example also uses pattern matching with multiple arguments, but this time the pattern matches occur within in the function definition – rather than a case expression . . . . .	22
24	This example extends the previous example. A new constructor <code>Consett</code> has been added to the data type <code>Weather</code> . The refactoring has matched the consistencies of the pattern matching used in the definition of <code>checkTemp</code> to create new pattern matches based on <code>Consett</code> . . . . .	23
25	This is an example where we may want to add a new constructor to a data type but there is a function definition which take multiple arguments, and both arguments are of the type in question.	24
26	This is an example where we may want to add a new constructor to a data type but there is a function definition which take multiple arguments, and both arguments are of the type in question. The refactoring again has tried to match the consistencies as best it could. . . . .	25
27	This is an example where we may want to add a new constructor to a data type but there is a function definition which take multiple arguments, and both arguments are of the type in question.	26
28	This is an example where one of the constructors of the data type in question is expressed as an infix operator . . . . .	28
29	Extending the previous example - a new constructor called <code>Sub</code> has been added to <code>Expr</code> . As a result, the refactoring has added a new pattern match to <code>calc</code> . The new pattern match has been defined with an infix operator because this new constructor was also defined as infix within the data type. It seems sensible to keep the consistencies of using the constructors as they were defined	28
30	This example shows where nested patterns could pose a problem. The next example shows when a constructor is added to the <code>Expr</code> data type . . . . .	29
31	This example shows nested patterns. A new constructor <code>Add</code> has been added to the data type <code>Expr</code> . The refactoring has added new pattern matching to <code>calc</code> and has tried to match the consistencies of the other patterns. . . . .	30

# Chapter 1

## Introduction

In this proposal I aim to outline the research that I intend to work on as part of my PhD research within the immediate future. I aim to implement a new program transformation for HaRe that allows a constructor to be added to a data type easily.

Creating a refactoring that automates the process of adding a constructor seems an obvious choice. After the constructor has been added to the data type, HaRe can then automatically generate new patterns for functions and case expressions which refer to the type in question. The next section looks at examples for adding a constructor to a data type more closely.

## Chapter 2

# Adding a Constructor to a Data Type

The aim of this refactoring is to allow the user to select a data type, select the “add constructor to data type” refactoring from the HaRe menu and then to input as a string the Constructor name followed immediately by the Type parameters that the constructor takes. Alternatively if the user wishes to enter the constructor as an infix expression, then they can also do this by typing in the type parameters and the constructor as an infix expression.

The refactoring then places this text directly into the file and relies upon the Programatica parser to convert the module with the added data type into an abstract representation and inform the user about any errors if they occur. HaRe will intercept the message returned by Programatica and report the error to the user in a more useful format.

Figure 2.1 below shows a simple example of how this refactoring could be used. The example shows a data type `Weather` with two constructors: `Canterbury` and `Newcastle`. The example also shows a function called `getTemp` which by definition takes a constructor from the `Weather` type and returns a `Temp` type. `getTemp` uses pattern matching over the type `Weather` to distinguish between different cases of the function `getTemp`.

Before looking at the example in figure 2.2, let us imagine that we want to add a new constructor to the data type `Weather` to include the town `Consett`. This change can be made manually but doing so would result in the user having to check where in the program the `Weather` data type is referenced so that the program will not fall over if the new constructor `Consett` is referred to.

It would be better to encompass this functionality into a data type refactoring and incorporate this refactoring into the HaRe project. Figure 2.2 shows the results of a refactoring that allows a constructor to be added to a data type.

There are a number of problems with this refactoring: Firstly there might already be a catch all clause within the pattern match (figure 2.3). There are two options to deal with this. The first option is to leave the function declaration as it when the new constructor has been added. A better solution would be to add a new pattern `getTemp (Consett t) = undefined` to the function `getTemp` (as seen in figure 2.4).

Another design problem is to consider the return type of the function in

```

module Temp1 where

data Temp = Hot | Cold
          deriving Show

data Weather = Canterbury Temp |
             Newcastle Temp
             deriving Show

getTemp :: Weather -> Temp
getTemp (Canterbury t) = t
getTemp (Newcastle t) = t

```

Figure 2.1: An example function for program slicing

```

module Temp1 where

{- the constructor Consett has been added
   and the refactoring will add the undefined clause to
   getTemp
-}

data Temp = Hot | Cold
          deriving Show

data Weather = Canterbury Temp |
             Newcastle Temp |
             Consett Temp
             deriving Show

-- would be nice to also add
addedConsett = error "Consett added to Weather"

getTemp :: Weather -> Temp
getTemp (Canterbury t) = t
getTemp (Newcastle t) = t

getTemp (Consett t) = addedConsett

```

Figure 2.2: An application of program applied to the sub-expression res

```

getTemp :: Weather -> Temp
getTemp (Canterbury t) = t
getTemp _ = undefined

```

Figure 2.3: There is already a catch all in the pattern match

```

getTemp :: Weather -> Temp
getTemp (Canterbury t) = t

getTemp (Consett t) = undefined

getTemp _ = undefined

```

Figure 2.4: There is already a catch all in the pattern match - added a new clause

```
tempToString :: Temp -> String
tempToString Hot = "It is hot!"
tempToString Cold = "It is freeezing!"
tempToString _ = undefined
```

Figure 2.5: An example showing a function created from the data dependencies

```
data Temp = Hot | Cold | Warm
  deriving Show

data Weather = Newcastle |
              Canterbury

data Month = Jan | Jun
```

Figure 2.6: A simple data type

question. Figure 2.5 shows a function which has a pattern match over the data type and returns a `String`. If a constructor `Consett` was added we have two choices. Firstly we could add a new clause to return the empty string (`''`) or we could create a new definition, `addedConsett=undefined` and then use the value of this definition for the new clause. It can be argued that returning a value of the same type as the function (in the case the empty string `''`) is more well defined then returning `undefined`. Similarly if the function returned a type `Int` then it might be feasible to return the case `0` for all clauses that are added as a result of adding the new pattern matches. The problem with this is that the refactoring does not know the intended purpose of this new constructor and predicting the return values of these new patterns could be wrong. It is more sensible to return `undefined`. The interesting problem is that one cannot always choose a sensible value for the function to return. `undefined` is not a sensible return value, but `undefined` is a conservative return value in that it does not try to guess the intended purpose of the function.

Some other design considerations are shown in Appendix A. Some interesting example in appendix A include:

- Testing if an error is raised when the same constructor name is declared in two places;
- Adding new patterns to a case expression rather than an equation pattern match;
- Cases where the modified type is a class instance;
- Cases where the modified data type uses file names.

Another problem to consider is if the equation pattern match uses multiple parameters. Consider the data type in figure 2.6, suppose we want to add a new constructor called `Consett` to the data type `Weather` – we would end up with a data type that is defined in figure 2.7. Now let us consider that we have a function `checkTemp` which has the type `Month -> Weather -> temp`,

```

data Temp = Hot | Cold | Warm
    deriving Show

data Weather = Newcastle |
              Canterbury |
              Consett

data Month = Jan | Jun

```

Figure 2.7: data type with a constructor added

```

checkTemp :: Month -> Weather -> Temp
checkTemp Jan Newcastle = Cold
checkTemp Jun Newcastle = Warm

checkTemp Jan Canterbury = Cold
checkTemp Jun Canterbury = Hot

```

Figure 2.8: A function which takes two data types as parameters

```

checkTemp :: Month -> Weather -> Temp
checkTemp Jan Newcastle = Cold
checkTemp Jun Newcastle = Warm

checkTemp Jan Canterbury = Cold
checkTemp Jun Canterbury = Hot

checkTemp Jan Consett = addedConsett
checkTemp Jun Consett = addedConsett

```

Figure 2.9: A function which takes two data types as parameters

```

checkTemp :: Month -> Weather -> Temp
checkTemp Jan Newcastle = Cold

checkTemp Jan Canterbury = Cold
checkTemp Jun Canterbury = Hot

checkTemp Jan Consett = addedConsett
checkTemp Jun Consett = addedConsett

```

Figure 2.10: A function which takes two data types as parameters

```
data Sign = Rock | Paper | Scissors
          deriving Show
```

Figure 2.11: The data type Sign

```
game :: Sign -> Sign -> Sign
game Rock Paper = Paper
game Rock Rock = Rock
game Rock Scissors = Rock
game Paper Scissors = Rock
game Paper Paper = Paper
game Scissors Scissors = Scissors.
```

Figure 2.12: A function game which works with two Sign parameters.

as shown in figure 2.8. If the constructor `Consett` is added to `Weather` then new pattern matching must be introduced into the definition of `checkTemp` to produce the result as shown in figure 2.9. The problem we face is if in the original definition, `checkTemp` does not pattern match every constructor of `Month` over every constructor in `Weather`. In this case the refactoring can only produce a new pattern match `checkTemp x y = undefined` as the refactoring cannot guess which constructors the user wants pattern matching over. Only when there are consistent pattern matches can the refactoring introduce new pattern matches based on these consistencies (e.g. every constructor in one type is matched with every constructor of another type). All that can be done in the cases where the pattern matches are inconsistent is to try and mimic the existing cases. This would mean that the refactoring would mimic pattern matches that may have been badly defined. Figure 2.10 shows an example of this.

Let us consider another example which requires additional work to what the other problems already discussed have outlined. Let us consider the data types in figure 2.11. Now suppose we also have a function called `game` which takes two `Sign` and determines the result of applying two `Signs` together (a `Rock` and `Scissors` would result in `Rock` blunting `Scissors`). The function `game` is defined in figure 2.12 (for purposes of brevity not all the cases have been included). The function `game` is a case where there are multiple parameters to the function, and both parameters are of the type in question. Now suppose a new constructor `Nuke` is added to `Game`, as defined in figure 2.13. This case follows the same argument before and the refactoring must look for consistencies in order to extend the pattern matching - otherwise the refactoring must produce the worst case pattern match. If both arguments are of the type in question

```
data Sign = Rock | Paper | Scissors | Nuke
          deriving Show
```

Figure 2.13: The data type Sign

then it can also be argued that the order of the parameters are of importance. In this case the refactoring must make a best guess at introducing new patterns and leave it up to the user to reorganise the ordering of the generated patterns.

Appendix C shows some interesting examples including nested patterns and infix constructors. It seems most sensible to reproduce a pattern that uses an infix constructor if and only if the constructor is declared as infix in the data type, it is important to keep consistencies.

# Appendix A - General cases

```

module Temp3 where

data Temp = Hot | Cold | Raining
          deriving Show

data Weather = Canterbury Temp |
              Newcastle Temp
             deriving Show

tempToString :: Temp -> String
tempToString Hot = "It is hot!"
tempToString Cold = "It is freeezing!"
tempToString _ = undefined

getTemp :: Weather -> String
getTemp (Canterbury t) = tempToString t
getTemp (Newcastle t) = tempToString t
getTemp _ = undefined

```

Figure 14: Shows a simple case where Newcastle has been added to Weather

```

module Temp3 where

data Temp = Hot | Cold | Raining
          deriving Show

data Weather = Canterbury Temp |
             Newcastle Temp |
             Consett Temp
             deriving Show

tempToString :: Temp -> String
tempToString Hot = "It is hot!"
tempToString Cold = "It is freeezing!"
tempToString _ = undefined

getTemp :: Weather -> String
getTemp (Canterbury t) = tempToString t
getTemp (Newcastle t) = tempToString t
getTemp _ = undefined

```

Figure 15: adding the constructor Consett to Weather in this example should result in programatica giving an error

```

module Temp4 where

data Temp = Hot | Cold | Raining
          deriving Show

data Weather = Canterbury Temp |
             Newcastle Temp
             deriving Show

tempToString :: Temp -> String
tempToString Hot = "It is hot!"
tempToString Cold = "It is freeezing!"
tempToString _ = undefined

getTemp :: Weather -> String
getTemp t = case t of
             (Newcastle t) -> tempToString t
             (Canterbury t) -> tempToString t

```

Figure 16: This file shows how getTemp works over the data type Weather the next figure shows a constructor being added to Weather

```

module Temp4 where

data Temp = Hot | Cold | Raining
          deriving Show

data Weather = Canterbury Temp |
              Newcastle Temp |
              Consett Temp
              deriving Show

addedConsett = error "constructor Consett added to Weather"

tempToString :: Temp -> String
tempToString Hot = "It is hot!"
tempToString Cold = "It is freeezing!"
tempToString _ = undefined

getTemp :: Weather -> String
getTemp t = case t of
  (Newcastle t) -> tempToString t
  (Canterbury t) -> tempToString t
  -- possible to add
  (Consett t) -> addedConsett
  - -> undefined

```

Figure 17: Consett has been added to Weather and the refactoring has added a wildcard to the function getTemp

```

module Temp4 where

class Print a where
  p :: a -> String

data Temp = Hot | Cold

instance Print Temp where
  p Hot = "It is hot!"
  p Cold = "It is cold"

data Weather = Canterbury Temp |
              Newcastle Temp |
              Consett Temp

getTemp :: Weather -> String
getTemp (Canterbury t) = p t
getTemp (Newcastle t) = p t
getTemp (Consett t) = p t

```

Figure 18: The refactoring also needs to take into account class instances, and pattern matches that occur within equations in the class instances

```

module Temp4 where

{-
-}

class Print a where
  p :: a -> String

data Temp = Hot | Cold | Raining

addedRaining = error "constructor Raining added to Temp"

instance Print Temp where
  p Hot = "It is hot!"
  p Cold = "It is cold"

  -- possible to add
  p Raining = addedRaining

  p _ = undefined

data Weather = Canterbury Temp |
              Newcastle Temp |
              Consett Temp

getTemp :: Weather -> String
getTemp (Canterbury t) = p t
getTemp (Newcastle t) = p t
getTemp (Consett t) = p t

```

Figure 19: The refactoring also needs to take into account class instances. The constructor Raining has been added to Temp and an undefined clause has been added to the Print instance for Temp

```

module Temp5 where

data Point = Pt {pointx, pointy :: Float}

abs' (Pt {pointx = x, pointy = y}) = sqrt (x*x + y*y)

module Temp5 where

-- added a new constructor with fields and the refactorer adds
-- an undefined clause to abs'

data Point = Pt {pointx, pointy :: Float} |
             PtR {pointx, pointy, pointz :: Float}

abs' (Pt {pointx = x, pointy = y}) = sqrt (x*x + y*y)
abs' _ = undefined

```

Figure 20: This example shows a case where the data type uses field names. In this case the refactorer should use field names if and only if they have been declared in the data type with field names.

# Appendix B - Further examples

```

module Data5 where

data Temp = Hot | Cold | Warm
    deriving Show

data Weather = Newcastle |
    Canterbury

-- only 2 months for an example.
data Month = Jan | Jun

checkTemp :: Month -> Weather -> Temp
checkTemp x Newcastle = case x of
    Jan -> Cold
    Jun -> Warm
checkTemp x Canterbury = case x of
    Jan -> Cold
    Jun -> Hot

```

Figure 21: This example shows where a case pattern matching is used with multiple arguments and only one of the arguments are of the type in question. The pattern matching matches every constructor from one data type with every constructor from the data type in question

```

module Data5 where

{- example to show that there is more than one pattern used for
   a function when the constructor Consett is added to Weather
-}

data Temp = Hot | Cold | Warm
    deriving Show

data Weather = Newcastle |
    Canterbury |
    Consett

addedConsett = error "constructor Consett added to Weather"

-- only 2 months for an example.
data Month = Jan | Jun

checkTemp :: Month -> Weather -> Temp
checkTemp x Newcastle = case x of
    Jan -> Cold
    Jun -> Warm
checkTemp x Canterbury = case x of
    Jan -> Cold
    Jun -> Hot
checkTemp x Consett = addedConsett

```

Figure 22: This example extends the case from the previous figure. A new constructor has been added to the data type `Weather`, the refactoring keeps the consistencies seen in the other function pattern matches but ignores the pattern matching in the case expression

```

module Data5 where

data Temp = Hot | Cold | Warm
    deriving Show

data Weather = Newcastle |
    Canterbury

-- only 2 months for an example.
data Month = Jan | Jun

checkTemp :: Month -> Weather -> Temp
checkTemp Jan Newcastle = Cold
checkTemp Jun Newcastle = Warm

checkTemp Jan Canterbury = Cold
checkTemp Jun Canterbury = Hot

```

Figure 23: This example also uses pattern matching with multiple arguments, but this time the pattern matches occur within in the function definition – rather than a case expression

```

module Data5 where

{- Consett has been added to Weather but checkTemp uses multiple patterns...
-}

data Temp = Hot | Cold | Warm
    deriving Show

data Weather = Newcastle |
    Canterbury |
    Consett

addedConsett = error "constructor Consett added to Weather"

-- only 2 months for an example.
data Month = Jan | Jun

checkTemp :: Month -> Weather -> Temp
checkTemp Jan Newcastle = Cold
checkTemp Jun Newcastle = Warm

checkTemp Jan Canterbury = Cold
checkTemp Jun Canterbury = Hot

checkTemp Jan Consett = addedConsett
checkTemp Jun Consett = addedConsett

```

Figure 24: This example extends the previous example. A new constructor `Consett` has been added to the data type `Weather`. The refactoring has matched the consistencies of the pattern matching used in the definition of `checkTemp` to create new pattern matches based on `Consett`.

```

module Data5 where

data Temp = Hot | Cold | Warm
    deriving Show

data City = Newcastle |
    Canterbury

-- only 2 months for an example.
data Month = Jan | Jun

checkTemp :: City -> Month -> Temp
checkTemp Newcastle Jan = Cold
checkTemp Newcastle Jun = Warm

checkTemp Canterbury Jan = Cold
checkTemp Canterbury Jun = Hot

module Data5 where

data Game = Win | Loose | Draw

data Sign = Rock | Paper | Scissors
    deriving Show

game :: Sign -> Sign -> Sign
game Rock Paper      = Paper
game Rock Rock       = Rock
game Rock Scissors   = Rock
game Paper Scissors  = Scissors
game Paper Paper     = Paper
game Scissors Scissors = Scissors

```

Figure 25: This is an example where we may want to add a new constructor to a data type but there is a function definition which take multiple arguments, and both arguments are of the type in question.

```

module Data5 where

{- shows where *two* arguments are of the type in question to a function

    nuclear warhead has been added to Sign

    In this example however, the ordering of the parameters is not important
-}

data Sign = Rock | Paper | Scissors | Nuke
          deriving Show

addedNuke = error "constructor Nuke has been added to Sign"

game :: Sign -> Sign -> Sign
game Rock Paper      = Paper
game Rock Rock       = Rock
game Rock Scissors   = Rock
game Paper Scissors  = Scissors
game Paper Paper     = Paper
game Scissors Scissors = Scissors

game Nuke Rock       = addedNuke
game Nuke Paper      = addedNuke
game Nuke Scissors   = addedNuke
game Nuke Nuke       = addedNuke

```

Figure 26: This is an example where we may want to add a new constructor to a data type but there is a function definition which take multiple arguments, and both arguments are of the type in question. The refactoring again has tried to match the consistencies as best it could.

```

module Data5 where

{- shows where *two* arguments are of the type in question to a function

    In this example the ordering of the 2 parameters affects the result
    of the function.
-}

data Sign = Rock | Paper | Scissors | Nuke
          deriving Show

addedNuke = error "constructor Nuke has been added to Sign"

game :: Sign -> Sign -> Sign
game Rock Paper      = Paper
game Rock Rock       = Rock
game Rock Scissors   = Rock
game Paper Scissors  = Scissors
game Paper Paper     = Paper
game Scissors Scissors = Scissors

game Nuke Rock       = addedNuke
game Nuke Paper      = addedNuke
game Nuke Scissors   = addedNuke
game Nuke Nuke       = addedNuke

```

Figure 27: This is an example where we may want to add a new constructor to a data type but there is a function definition which take multiple arguments, and both arguments are of the type in question.

# Appendix C - Extended examples

```

module Data5 where

{- this module incorporates infix data types
-}

data Expr = Lit Int |
           Expr 'Add' Expr
           deriving (Show)

calc :: Expr -> Int
calc (Lit x) = x
calc (x 'Add' y) = (calc x) + (calc y)

```

Figure 28: This is an example where one of the constructors of the data type in question is expressed as an infix operator

```

module Data5 where

{- this module incorporates infix data types

   Sub has been added to Expr as an infix constructor
-}

data Expr = Lit Int |
           Expr 'Add' Expr |
           Expr 'Sub' Expr
           deriving (Show)

addedExpr = error "added Sub to Expr"

calc :: Expr -> Int
calc (Lit x) = x
calc (x 'Add' y) = (calc x) + (calc y)
calc (x 'Sub' y) = addedExpr

```

Figure 29: Extending the previous example - a new constructor called `Sub` has been added to `Expr`. As a result, the refactoring has added a new pattern match to `calc`. The new pattern match has been defined with an infix operator because this new constructor was also defined as infix within the data type. It seems sensible to keep the consistencies of using the constructors as they were defined

```
module Data5 where

{-
    this shows nested patterns
-}

data Pats = Pat1 Expr Expr |
           Pat2 Expr

data Expr = Lit Int

calc :: Pats -> Int
calc (Pat1 (Lit x) (Lit y)) = x + y
calc (Pat2 (Lit x)) = x
```

Figure 30: This example shows where nested patterns could pose a problem. The next example shows when a constructor is added to the `Expr` data type

```

module Data5 where

{-
    this shows nested patterns
-}

data Pats = Pat1 Expr Expr |
           Pat2 Expr
           deriving Show

data Expr = Lit Int |
           Add Expr Expr
           deriving Show

addedAdd = error "added Add to Expr"

calc :: Pats -> Int
calc (Pat1 (Lit x) (Lit y)) = x + y
calc (Pat2 (Lit x)) = x

-- a possible way...
calc (Pat1 (Add x y) (Lit z)) = addedAdd
calc (Pat1 (Lit x) (Add y z)) = addedAdd
calc (Pat2 (Add x y)) = addedAdd

```

Figure 31: This example shows nested patterns. A new constructor `Add` has been added to the data type `Expr`. The refactoring has added new pattern matching to `calc` and has tried to match the consistencies of the other patterns.

```

module Data5 where

{-
  Extended ex28 - this time a constructor is added to Pats
  this shows nested patterns
-}

data Pats = Pat1 Expr Expr |
           Pat2 Expr      |
           Pat3 Expr Expr Expr
           deriving Show

addedPat3 = error "added Pat3 to Pats"

data Expr = Lit Int |
          Add Expr Expr
          deriving Show

addedAdd = error "added Add to Expr"

calc :: Pats -> Int
calc (Pat1 (Lit x) (Lit y)) = x + y
calc (Pat2 (Lit x)) = x

-- a possible way...
calc (Pat1 (Add x y) (Lit z)) = addedAdd
calc (Pat1 (Lit x) (Add y z)) = addedAdd
calc (Pat2 (Add x y)) = addedAdd

calc (Pat3 (Lit a) (Lit b) (Lit c)) = addedPat3
calc (Pat3 (Lit a) (Add b c) (Lit d)) = addedPat3
calc (Pat3 (Lit a) (Lit b) (Add c d)) = addedPat3
calc (Pat3 (Add a b) (Lit c) (Lit d)) = addedPat3
calc (Pat3 (Lit a) (Add b c) (Add d e)) = addedPat3
calc (Pat3 (Add a b) (Lit c) (Add d e)) = addedPat3
calc (Pat3 (Lit a) (Add b c) (Add d e)) = addedPat3

```

Figure 32: A rather interesting example. This shows a case where there is an extensive use of pattern matching. The constructor `Pat3` has been added to the data type `Pats` and the refactoring has noticed that the consistency seems to be to nest the data type `Expr` within every case of `Pats`. New pattern matches have been added to match this consistency