Language: Haskell

Headline

The functional programming language Haskell

Details

101wiki hosts plenty of Haskell-based contributions. This is evident from corresponding back-links. More selective sets of Haskell-based contributions are organized in themes: Theme:Haskell data, Theme:Haskell potpourri, and Theme:Haskell genericity. Haskell is also the language of choice for a course supported by 101wiki: Course:Lambdas_in_Koblenz.

Illustration

The following expression takes the first 42 elements of the infinite list of natural numbers:

```
> take 42 [0..]
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41]
```

In this example, we leverage Haskell's lazy evaluation.

Metadata

- http://www.haskell.org/
- http://en.wikipedia.org/wiki/Haskell_(programming_language)
- Functional programming language

Concept: Functional programming language

Headline

A programming language for functional programming

Metadata

- Programming language
- Functional programming

Course: Lambdas in Koblenz

Headline

Introduction to functional programming at the University of Koblenz-Landau

Schedule of latest edition

- Lecture First steps
- Lecture Basic software engineering
- Lecture Searching and sorting
- Lecture Basic data modeling
- Lecture Higher-order functions
- Lecture Type-class polymorphism
- Lecture Functors and friends
- Lecture Functional data structures
- Lecture Unparsing and parsing

Additional lectures

- Lecture Monads
- Lecture Generic functions

Metadata

- http://softlang.wikidot.com/course:fp

Concept: Lazy evaluation

Headline

Delay evaluation of an expression until its value is needed

Illustration

Lazy by definition

Lazy evaluation is either supported by the underlying programming language or it needs to be encoded by the programs. Let's start with illustrations in Haskell; this language's semantics is lazy by definition.

Consider the following expression and its evaluation:

```
> repeat 42
[42,42,42,42,42,42,42,42,42,42,42,42,42,42,...
```

That is, 42 is to be repeated an infinite number of times and all those 42s are to be collected in one list. It is not surprising that the evaluation of this expression never stops as witnessed by printing the infinite result forever. Laziness comes into play when such expressions are used in a way that they do not need to be fully evaluated.

For instance, let us compute the head of an infinite list:

```
> head $ repeat 42
42
```

Thus, the list of repeated 42s is never materialized; rather the infinite list is only computed up to the point needed for returning the result, i.e., the head of the list. Here is another example for exploiting laziness to compute on 'infinite' data:

```
> length $ take 42 $ repeat 42
42
```

That is, we compute the length of the list that holds the first 42 elements of the earlier infinite list of 42s. Here is yet another example:

```
> [1..] !! 41
42
```

That is, we retrieve the 42nd element (the 41st index) of the earlier list.

Lazy conditionals

Most languages are readily lazy in terms of the semantics of their conditionals such that the 'then' and 'else' branches are only evaluated or executed, if necessary. This specific form of

laziness is obviously important for programming, regardless of whether we face a language with lazy or strict evaluation. For instance, consider the following definition of factorial in Haskell:

```
-- A straightforward definition of factorial
factorial :: Integer -> Integer
factorial x =
  if x < 0
    then error "factorial arg error"
    else if x <= 1
      then 1
      else x * factorial (x-1)
```

Regardless of language, such a definition should not evaluate the recursive case, except when honored by the value of the condition. Thus, this style of recursive definition even works in a programming language with strict evaluation, .e.g, in Python:

```
# A straightforward definition of factorial
def factorial(x):
    if not isinstance(x, (int, long)) or x<0:
        raise RuntimeError('factorial arg error')
    else:
        if x <= 1:
            return 1
        else:
            return x * factorial(x-1)
```

The difference between lazy and eager evaluation becomes quite clear, when we attempt a definition of 'if' as a function. In Haskell, we can actually define a function to mimic 'if' and use it in revising the recursive definition of factorial:

```
-- A re-definition of "if"
ifThenElse :: Bool -> x -> x -> x
ifThenElse True x   = x
ifThenElse False   x = x

-- Factorial re-defined to use user-defined if
factorial' :: Integer -> Integer
factorial' x =
  ifThenElse (x < 0)
    (error "factorial arg error")
    (ifThenElse (x <= 1)
      1
      (x * factorial' (x-1)))
```

The fact that this definition works depends on the lazy evaluation semantics of Haskell. The arguments of the function *ifThenElse* are only evaluated, when they are really needed. Let us attempt the same experiment in a language with eager evaluation semantics, e.g., Python:

```
# A troubled re-definition of "if"
def troubledIf(b,x1,x2):
    if b:
        return x1
    else:
        return x2

# Factorial re-defined to use user-defined if
def troubledFactorial(x):
```

```
    if not isinstance(x, (int, long)) or x<0:
        raise RuntimeError('factorial arg error')
    else:
        return troubledIf(x<=1,1,x * troubledFactorial(x-1))
```

When exercising this definition, we get this sort of runtime error:

```
>>> troubledFactorial(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "program.py", line 23, in troubledFactorial
    return troubledIf(x<=1,1,x * troubledFactorial(x-1))
  File "program.py", line 23, in troubledFactorial
    return troubledIf(x<=1,1,x * troubledFactorial(x-1))
  File "program.py", line 23, in troubledFactorial
    return troubledIf(x<=1,1,x * troubledFactorial(x-1))
  File "program.py", line 23, in troubledFactorial
    return troubledIf(x<=1,1,x * troubledFactorial(x-1))
  File "program.py", line 23, in troubledFactorial
    return troubledIf(x<=1,1,x * troubledFactorial(x-1))
  File "program.py", line 23, in troubledFactorial
    return troubledIf(x<=1,1,x * troubledFactorial(x-1))
  File "program.py", line 21, in troubledFactorial
    raise RuntimeError('factorial arg error')
RuntimeError: factorial arg error
```

A quick analysis suggests that this runtime error arises from the fact that an application of the function 'generates' an infinite chain of recursive applications, thereby eventually leading to the application of the function to a negative number, which is intercepted by the precondition test of the function. Thus, the function *troubledIf* is clearly not lazy and it cannot be used in defining the factorial function.

Encoding laziness

One may encode laziness in a language with eager evaluation. To this end, each expressions, for which evaluation should be deferred, can be turned into a degenerated closure (lambda abstraction) such that the evaluation can be requested explicitly by a trivial application. Consider the following attempt at a user-defined 'if' in Python and its use in another attempt at the factorial function:

```
# A (properly) lazy re-definition of "if"
def lazyIf(b,x1,x2):
    if b:
        return x1(())
    else:
        return x2(())

# A definition of factorial using lazyIf
def lazyFactorial(x):
    if not isinstance(x, (int, long)) or x<0:
        raise RuntimeError('factorial arg error')
    else:
        return lazyIf(x<=1,lambda  : 1, lambda  : x * lazyFactorial(x-1))
```

Thus, evaluation is requested explicitly by passing "()" (i.e., the empty tuple) to a "deferred" expression. When constructing a deferred expression, then we use a lambda abstraction with a

superfluous variable.

See Document:Okasaki96 for a profound discussion of data structures in a functional programming language while leveraging laziness for the benefit of efficiency.

Relationships

See the related concept of eager evaluation.

Synonyms (in a broad sense):

- Call-by-need evaluation
- Non-strict evaluation
- Laziness

Metadata

- http://en.wikipedia.org/wiki/Lazy evaluation
- Evaluation strategy
- Vocabulary:Programming
- Vocabulary:Programming languages

Theme: Haskell data

Headline

Varying Haskell-based approaches to data modeling

Description

Different feature models and design choices are exercised for theLanguage:Haskell-based data model of companies. Thereby, Haskell's data modeling expressiveness and common styles are explored. Here is summary of included contributions and reasons for inclusion:

- haskellStarter: A data model with only type synonyms
- haskellData: A data model with algebraic data types
- haskellRecord: A data model with record types
- haskellComposition: A data model with composition
- haskellVariation: A data model with variation
- haskellTermRep: A universal term representation

Any mentioning of "trivial data model" implies Feature:Flat company as opposed to Feature:Hierarchical_company. The remaining contributions involve data models that deal with Feature:Hierarchical_company. It should be noted that the contributions may serve additional purposes other than just illustrating data modeling options.

Relationships

There are further themes with Haskell-based contributions:

- Theme:Haskell introduction: basics of Haskell.
- Theme:Haskell potpourri: more advanced and diverse coverage of the Haskell ecosystem.
- Theme:Haskell genericity: different styles of generic functional programming in Haskell.

Metadata

Theme: Haskell genericity

Headline

Varying generic programming approaches in Haskell

Description

There are different classes of generic programming. The present theme is concerned with the class of generic programming that involves data type-polymorphic functions such that the functions can be applied to data of different types as, for example, in the case of the "Scrap your boilerplate" style of generic programming. The present theme is focused on different generic programming styles as they exist for Language:Haskell. Certain features of the system:Company are particularly relevant for the present theme. These are the features for cutting and totaling salaries as they illustrate the need for data transformations and queries that may need to fully traverse compound data while only some details of such data (i.e., salaries) are conceptually relevant. Thus, Feature:Total and Feature:Cut make up the baseline set of features to be covered by any member contribution of this theme.

These are the members of the theme:

- haskellSyb: "Scrap your boilerplate" style
- strafunski: Strategic programming
- haskellTree: Rose trees for representation
- tabaluga: Folds for systems of data types

Relationships

- See Theme:Scrap your boilerplate for a specific style of generic programming with Haskell coverage.

Metadata

- Feature:Total
- Feature:Cut
-

Theme: Haskell potpourri

Headline

A potpourri of Haskell-based contributions

Description

This theme demonstrates Language:Haskell's approach to several programming problems: concurrent programming, database programming, generic programming, GUI programming, logging, parsing, unparsing, XML programming, web programming. Some of the contributions nicely demonstrate some strengths and specifics of Haskell. This is true, arguably, for the contributions that illustrate XML programming and generic programming. Some other contributions are mainly included to provide coverage for important programming domains or problems without necessarily arguing that the Haskell-based approach is particularly interesting or attractive. This is true, for example, for the contribution that demonstrates GUI programming. Relatively mature and established technologies are demonstrated as opposed to research experiments.

The theme collects the following Haskell-based contributions:

- haskellParsec: Parsing with the Parsec library
- hughesPJ: Unparsing with Text.PrettyPrint.HughesPJ
- wxHaskell: GUI programming with wxHaskell
- happstack: Web programming with Happstack
- haskellDB: Database programming with HaskellDB
- hxt: XML programming with HXT
- writerMonad: Logging with the writer monad
- mvar: Concurrent programming with MVars
- haskellSyb: Generic programming in SYB style

Relationships

There are further themes with Haskell-based contributions:

- Theme:Haskell introduction: basics of Haskell.
- Theme:Haskell data: mostly simply variations on data modeling in Haskell.
- Theme:Haskell genericity: different styles of generic functional programming in Haskell.

Metadata

-
