Evaluation Strategies

Call-me-maybe

Moritz Flucht

Institute for Software Engineering and Programming Languages
Universität zu Lübeck

December 7th, 2015

Concepts of Programming Languages

Motivation

"At a conference on programming languages you might hear someone say, 'The normal-order language Hassle has certain strict primitives. Other procedures take their arguments by lazy evaluation.'"

— H. Abelson,

The Structure and Interpretation of Computer Programs

Contents

Taxonomy

Semantics vs. Evaluators

Describing programming languages

Applicative-order

All parameters are evaluated **before** entering the body.

Found in traditional languages like *C, Java, Ruby*.

Normal-order

Parameter evaluation is deferred until they are **used**.

Found in functional languages like *Haskell and Miranda*.

Semantics vs. Evaluators

Describing programming languages

Example: try-function in Scheme

Consider

```
(define (try a b) (if (= a 0) 1 b))
```

being called with `(try 0 (/ 42 0))`

Semantics vs. Evaluators

Describing procedures and parameters

Strictness

A strict parameter is evaluated **before** the function is applied.

Read: a function x is strict in its parameter y.

Non-Strictness

A non-strict parameter is evaluated at **first use**.

Semantics vs. Evaluators

Describing evaluators

Eager evaluation

Expressions are evaluated when they are **encountered**.

Lazy evaluation

Expressions are evaluated when they are **needed**.

Lazy evaluation

Lazy evaluation

Deferring expression evaluation

- Class of strategies, not one single technique
- Delay evaluation to the last possible moment
- Unused expressions might never be evaluated

Example: unless in lazy Scheme

```
1  (define (unless condition _then _else)
2    (if condition _else _then))
```

Lazy evaluation

Call-by-name vs. call-by-need

Call-by-name

- Expressions are evaluated **every time** their value is needed
- Similar to macro expansion

Call-by-need

- The computed value is stored after first usage
- Call-by-name with **memoization**
- Only practical with immutable values and pure functions

Lazy evaluation

Why bother?—Advantages of laziness

- More concise and intuitive code (have laziness take care of the fallout)
- Generate-and-filter paradigm (intermediate lists)
- Infinite data structures

Example: Infinite lists in Haskell

Consider calling functions like `length`, `sort` and `doubleList` on

```
1  magic :: Int -> Int -> [Int]
2  magic m n = m : (magic n (m+n))
```

Lazy evaluation

Why bother?—Advantages of laziness

- `getIt (magic 1 1) 3`

```
1  getIt :: [Int] -> Int -> Int
2  getIt []     _ = undefined
3  getIt (x:xs) 1 = x
4  getIt (x:xs) n = getIt xs (n-1)
```

Lazy evaluation

Why bother?—Advantages of laziness

- `getIt (magic 1 1) 3`
- `getIt (1 : (magic 1 (1+1))) 3`

```
1  getIt :: [Int] -> Int -> Int
2  getIt []     _ = undefined
3  getIt (x:xs) 1 = x
4  getIt (x:xs) n = getIt xs (n-1)
```

Lazy evaluation

Why bother?—Advantages of laziness

```haskell
getIt :: [Int] -> Int -> Int
getIt []     _ = undefined
getIt (x:xs) 1 = x
getIt (x:xs) n = getIt xs (n-1)
```

- `getIt (magic 1 1) 3`
- `getIt (1 : (magic 1 (1+1))) 3`
- `getIt (magic 1 (1+1)) (3-1)`

Lazy evaluation

Why bother?—Advantages of laziness

```
1  getIt :: [Int] -> Int -> Int
2  getIt []     _ = undefined
3  getIt (x:xs) 1 = x
4  getIt (x:xs) n = getIt xs (n-1)
```

- `getIt (magic 1 1) 3`
- `getIt (1 : (magic 1 (1+1))) 3`
- `getIt (magic 1 (1+1)) (3-1)`
- `getIt (1 : (magic (1+1) (1+(1+1)))) (3-1)`

Lazy evaluation

Why bother?—Advantages of laziness

```haskell
getIt :: [Int] -> Int -> Int
getIt []     _ = undefined
getIt (x:xs) 1 = x
getIt (x:xs) n = getIt xs (n-1)
```

- `getIt (magic 1 1) 3`
- `getIt (1 : (magic 1 (1+1))) 3`
- `getIt (magic 1 (1+1)) (3-1)`
- `getIt (1 : (magic (1+1) (1+(1+1)))) (3-1)`
- `getIt (1 : (magic (1+1) (1+(1+1)))) 2`

Lazy evaluation

Why bother?—Advantages of laziness

```haskell
getIt :: [Int] -> Int -> Int
getIt []     _ = undefined
getIt (x:xs) 1 = x
getIt (x:xs) n = getIt xs (n-1)
```

- `getIt (magic 1 1) 3`
- `getIt (1 : (magic 1 (1+1))) 3`
- `getIt (magic 1 (1+1)) (3-1)`
- `getIt (1 : (magic (1+1) (1+(1+1)))) (3-1)`
- `getIt (1 : (magic (1+1) (1+(1+1)))) 2`
- `getIt (magic (1+1) (1+(1+1))) (2-1)`

Lazy evaluation

Why bother?—Advantages of laziness

```
1  getIt :: [Int] -> Int -> Int
2  getIt []     _ = undefined
3  getIt (x:xs) 1 = x
4  getIt (x:xs) n = getIt xs (n-1)
```

- `getIt (magic 1 1) 3`
- `getIt (1 : (magic 1 (1+1))) 3`
- `getIt (magic 1 (1+1)) (3-1)`
- `getIt (1 : (magic (1+1) (1+(1+1)))) (3-1)`
- `getIt (1 : (magic (1+1) (1+(1+1)))) 2`
- `getIt (magic (1+1) (1+(1+1))) (2-1)`
- `getIt (magic 2 (1+2)) (2-1)`

Lazy evaluation

Why bother?—Advantages of laziness

```haskell
getIt :: [Int] -> Int -> Int
getIt []     _ = undefined
getIt (x:xs) 1 = x
getIt (x:xs) n = getIt xs (n-1)
```

- `getIt (magic 1 1) 3`
- `getIt (1 : (magic 1 (1+1))) 3`
- `getIt (magic 1 (1+1)) (3-1)`
- `getIt (1 : (magic (1+1) (1+(1+1)))) (3-1)`
- `getIt (1 : (magic (1+1) (1+(1+1)))) 2`
- `getIt (magic (1+1) (1+(1+1))) (2-1)`
- `getIt (magic 2 (1+2)) (2-1)`
- `getIt (2 : magic (1+2) (2+(1+2))) (2-1)`

Lazy evaluation

Why bother?—Advantages of laziness

```haskell
getIt :: [Int] -> Int -> Int
getIt []     _ = undefined
getIt (x:xs) 1 = x
getIt (x:xs) n = getIt xs (n-1)
```

- `getIt (magic 1 1) 3`
- `getIt (1 : (magic 1 (1+1))) 3`
- `getIt (magic 1 (1+1)) (3-1)`
- `getIt (1 : (magic (1+1) (1+(1+1)))) (3-1)`
- `getIt (1 : (magic (1+1) (1+(1+1)))) 2`
- `getIt (magic (1+1) (1+(1+1))) (2-1)`
- `getIt (magic 2 (1+2)) (2-1)`
- `getIt (2 : magic (1+2) (2+(1+2))) (2-1)`
- `getIt (2 : (magic (1+2) (2+(1+2))) 1`

Lazy evaluation

Why bother?—Advantages of laziness

```haskell
1 getIt :: [Int] -> Int -> Int
2 getIt []     _ = undefined
3 getIt (x:xs) 1 = x
4 getIt (x:xs) n = getIt xs (n-1)
```

- `getIt (magic 1 1) 3`
- `getIt (1 : (magic 1 (1+1))) 3`
- `getIt (magic 1 (1+1)) (3-1)`
- `getIt (1 : (magic (1+1) (1+(1+1)))) (3-1)`
- `getIt (1 : (magic (1+1) (1+(1+1)))) 2`
- `getIt (magic (1+1) (1+(1+1))) (2-1)`
- `getIt (magic 2 (1+2)) (2-1)`
- `getIt (2 : magic (1+2) (2+(1+2))) (2-1)`
- `getIt (2 : (magic (1+2) (2+(1+2))) 1`
- `2`

Lazy evaluation

Implementation: think, thought, *thunk*

- Thunks are allocated to store an unevaluated expression
- Keep track of evaluation status and the environment
- Can be *forced* if a value is needed
  - Passing to a primitive function
  - Use in a condition
  - Used as an operation

Lazy evaluation

Implementation: think, thought, *thunk*

Example: Thunk representations in Python

```
1  class Thunk:
2    def init (self, expr, env):
3      self._expr = expr
4      self._env = env
5      self._evaluated = False
6    def value(self):
7      if not self._evaluated:
8        self._value = force_eval(self._expr, self._env)
9        self._evaluated = True
10     return self._value
```

Lazy evaluation

The downside

- Thunk allocation costs and bookkeeping
- Complicating error handling (order of execution)
- Impractical for mutable state (“lazy + state = death”)
- Extensive optimizations necessary
  - Data and control flow analysis
  - strictness analysis to determine what is *always evaluated*
  - cheap eagerness to speculate *low cost* expressions

Eager evaluation

Call-by-value vs. call-by-sharing

"This is a precise description of a fuzzy mechanism."

Expressions are evaluated **before** the application of a function.

Call-by-value

- Semantic implications for the perception of changes (scope)
- Creates a copy of a value—usually for primitive types

Call-by-sharing

- Maybe technically equivalent to call-by-value
- Changes to parameters potentially observable
- Requires mutable values to be distinguished

Lazy features in applicative-order languages

- Special forms are already present (e.g., if-clause)
- Short circuit evaluation as a compiler optimization
- Callbacks, promises and futures
- Call-by-name or call-by-need offered for certain value or function definitions on an opt-in basis

Lazy features in applicative-order languages

Example: Lazy pipelining in JavaScript

Pipelining in JavaScript with lodash was recently patched to behave lazily

```
1  function ageLt(x) {
2      return function(person) {
3          return person.age < x; };
4  }
5  var people = [
6      { name: 'Anders', age:  40 }, { name: 'Binca', age: 150},
7      { name: 'Conrad', age: 200 }, { name: 'Dieta', age:  70},
8      { name: 'Finnja', age:  30 }, { name: 'Geras', age: 130},
9      { name: 'Hastig', age:  20 }, { name: 'Ickey', age: 200}];
10 var alive = _(people)
11     .filter(ageLt(100)).take(3).value();
```

Optimistic evaluation

The concept of optimistic evaluation

- Use on top of already mentioned optimization
- Many thunks are “always used” or “cheap and usually used”
- Speculatively evaluate these thunks using call-by-value
- Abort if the computation takes too long

Compiler modifications

- These optimizations are added to the back end of the compiler
- Code generation for the intermediate language

let expressions

For every **let** expression `let x = <rhs> in <body>` we generate a switch

```
1  if (LET42 != 0) {
2      x = value of <rhs>
3  } else {
4      x = lazy thunk to compute <rhs>
5            when needed
6  } evaluate <body>
```

Compiler modifications

Abortion

- Initially, assume all thunks are sepculated
- *Abort* thunks with too long a computation
- Measured by sample points in an online profiling component

Chunky evaluation

- Deal with self-referencing data structures and lists
- Implemented by means of a recursion counter
- Switch to lazy evaluation after a given number of speculations

Compiler modifications

Additional things to consider:

- Errors in speculated evaluation must not be raised (yet)

Compiler modifications

Additional things to consider:

- Errors in speculated evaluation must not be raised (yet)
- Unsafe operations have to be excluded from speculation (I/O)

Run-time component & Results

- Switches can be adjusted during run-time
- Speculations are profiled by means of a heuristic (wastage quotient)
- Does the potentially wasted work outweigh the cost of thunk allocation?
- Optimistic evaluation produced a mean speed-up of 15% in test environment

Conclusion

Conclusion

Evaluator	Language	Procedures & parameters	Strategies
eager	applicative-order	strict	call-by-value, call-by-sharing
lazy	normal-order	non-strict	call-by-name, call-by-need

Conclusion

Evaluator	Language	Procedures & parameters	Strategies
eager	applicative-order	strict	call-by-value, call-by-sharing
lazy	normal-order	non-strict	call-by-name, call-by-need

- Optimizations to be made in both eager and lazy
- Choice dependent on use case, good to have both