TDDA69 Data and Program Structure

Declarative Computation Model

Cyrille Berger

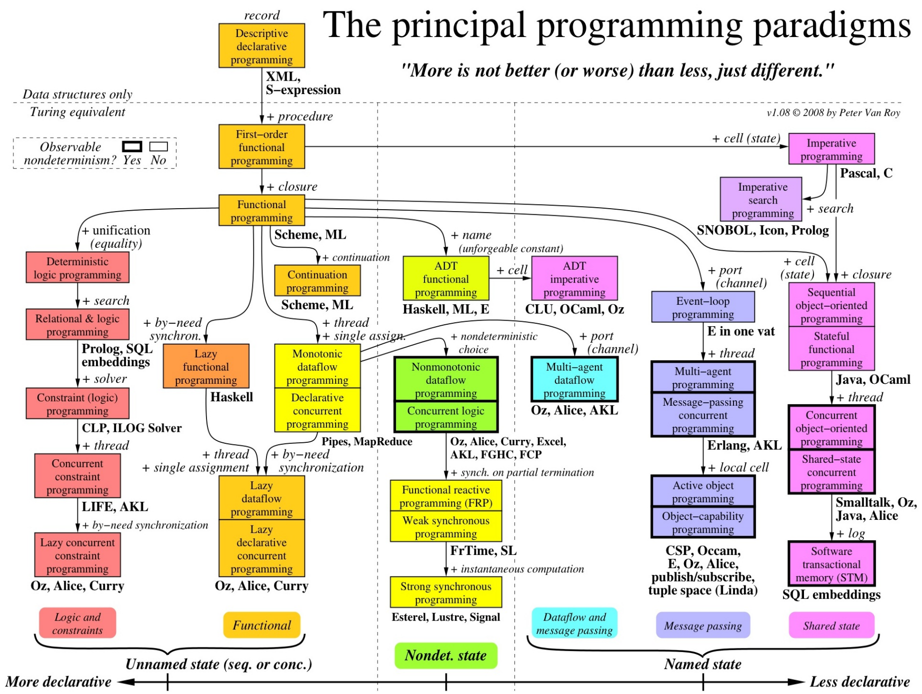
LINKÖPING UNIVERSITY

Lectures

Lecture content

- Declarative programming
- Evaluation model
  - Single Assignment Store
- Functional programming

Declarative programming



Declarative

- Expresses logic of computation without control flow:
  - What should be computed and not how it should be computed.
- Examples: XML/HTML, antlr4/yacc/ regular expressions, make/ants, SQL, ...

Declarative concepts

- First-order functional programming
  - record

    association of a name to a value
  - procedure

    a set of statements that can be called with a set of arguments

Minimum declarative syntax

- STATEMENT := (BLOCK | DEFINITION | CALL | ASSIGNMENT) ';'
- BLOCK := '{' STATEMENT* '}'
- DEFINITION := 'define' IDENTIFIER = EXPR | CALL | FUNCTION
- CALL := IDENTIFIER '(' (EXPR (, EXPR)*)) | () ')'
- FUNCTION := 'function' '(' (EXPR (, EXPR)+)) | () ')' ('->' IDENTIFIER) BLOCK
- EXPR := IDENTIFIER | NUMBER
- ASSIGNMENT := IDENTIFIER '=' EXPR

Example of minimum declarative syntax

```
define v = add(1, 3);
define f1 = function (x) -> y
{
  y = mul(x, x);
}
define v2;
v2 = f1(v);
```


Predefined functions

- Arithmetic: add, sub, mul, div...
- Control: cond

  cond(test, value, elsevalue)

  ```
  cond(x == 0, 0, 1 / x)
  ```


Evaluation model

Substitution Model - Example (1/2)

```
define square = function (x) -> xx { xx = mul(x, x);
};
define sum_of_squares = function(x, y) -> xxyy {
  define xx = square(x);
  define yy = square(y);
  xxyy = add(xx, yy);
};
define f = function(a) -> v {
  define a1 = add(a, 1);
  define a2 = mul(a, 2);
  v = sum_of_squares(a1, a2);
};
```

- How to evaluate:

  ```
  define a=add(4, 1);
  f(a)
  ```

Substitution Model – Example (2/

- To evaluate:
  `f(a)`
- Evaluate the arguments
  `f(5)`
- Fetch the value of *f*

```
define f = function(a) -> v {
   define a1 = add(a, 1);
   define a2 = mul(a, 2);
   v = sum_of_squares(a1, a2);
};
```

- Replace *a* by *5*

```
define f = function(a) -> v {
   define a1 = add(5, 1);
   define a2 = mul(5, 2);
   v = sum_of_squares(a1, a2);
};
```

- Evaluate the

```
define a1 = 6;
  define a2 = 10;
```

- Evaluate the

```
v = sum_of_squares(6, 10)
```

- Fetch the value of *sum_of_squares* and replace the arguments:

```
define xx = square(6);
  define yy = square(10);
  xxyy = add(xx, yy);
```

- And so

```
xx = 36;
yy = 100
xxyy = add(36, 100)
136
```

- This is the substitution model in the applicative order

Applicative vs Normal

- Recursively evaluate all the subexpressions, but in what order?
  - *Applicative order*: evaluate the arguments and then apply. Evaluate the procedure body with every formal parameter replaced with the corresponding argument value
  - *Normal order*: fully expand and then reduce. Recursively expand every procedure until only primitive operators are left. Evaluate (reduce) the result.

Applicative vs Normal – Example (1/

```
define square = function (x) ->
xx { xx = mul(x, x); };
```

- Applicative order

```
define v = mul(4,2);
square(v)
square(8))
mul(8, 8)
64
```

- Normal order

```
define v = mul(4,2);
square(v)
muul(mul(4, 2), mul(4,
2))
mul(8, 8)
64
```

For procedure applications that can be modeled with the substitution model that terminate with legitimate values, applicative and normal order produce the same value.

Applicative vs Normal – Example (2/

```
define foo = function (x, y) -
> z {
  z = cond(x == 0, y, mul(2,
x));
```

- Applicative order

```
define v =
div(20, 0);
   foo(10, v);
```

error: division by zero

- Normal order

```
define v = div(20,
0);
   foo(10, v);
z = cond(10 == 0,
div(20, 0), mul(2,
10));
mul(2, 10);
20
```

What is the result?

Normal order and lazy evaluation

In practice, COND is usually special cased

```
if(object and object.count()
              and object.sum() /
                             object.(
> 0.90):
  print('High success rate!')
```


Single Assignment Store

Assignement

- Binds names to values
- a := 2

  Now *a* has the value *2*
- a plus 2

  evaluates to *4*
- How to keep track of the values?

Single Assignement Store (1/3

- A single assignment store is a store (set) of variables
- Initially the variables are unbound, i.e. do not have a defined value
- Example: a store with three variables

```
define x;
define y;
define z;
```

store

x: unbound
y: unbound
z: unbound

Single Assignement Store (2/3

- Variables may be bound to values
- Example: a store with three variables
- Binding is done with a single assignment operation

```
x = 314;
y = [1, 2, 3];
```

store

x: 314
y: [1, 2, 3]
z: unbound

Single Assignement Store (3/3

- A declarative variables starts *unbound*
- It can be *bound* to exactly **one**value
- Once bound it stays bound through the computation, and is indistinguishable from its value

```
x = 314;
x = 414; // Error!
```

store

x: 314
y: [1, 2, 3]
z: unbound

Value Store

- A store where all variables are bound to values is called a value store
- This is enough for most functional programming languages

store

x: 314
y: [1, 2, 3]
z: 'Hello'

Single Assignment Store and Functio

- Global store

```
define x = 10;
```

- Each function has its own store

```
define f = function (a,
b) -> r {
   r = add(a, b);
}
```

- Arguments are binded in the value store

```
f(2, 3);
```

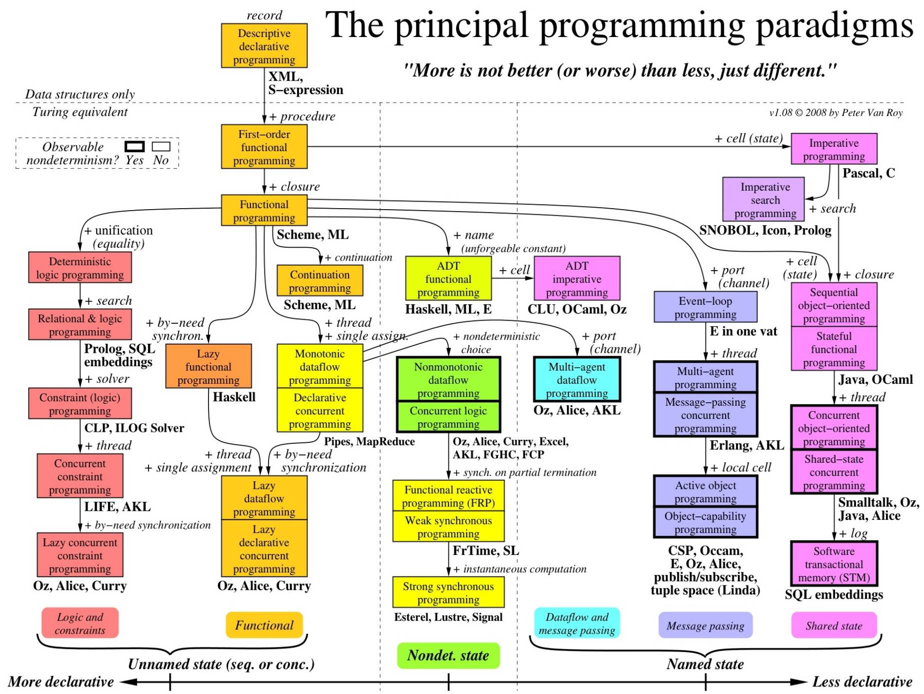
*(global)***I**

x:10

*(f)***II**

a: 2
b: 3
r: 5

Functional programming



Procedure vs Closure

- Procedure
  - Access only to its own store
- Closure
  - Like a procedure but with references to externally stored values

Single Assignment Store and Closure (1/

- Global store
  ```
  define x = 10;
  ```
- A closure has its own store with a copy of the store where it is created
  ```
  define f = function (a)
  -> r {
    r = add(a, x);
  }
  ```
- When calling the function
  ```
  f(2);
  ```

(global) **I**
x:10

(f) **II**
x:10
a: 2
r: 12

Single Assignment Store and Closure (3/

```
define x = 10;
define make_f =
  function (a) -> r
{
    r = function(b)
-> s {
      s =  add(a,
mul(x, b));
    };
  }
define f =
make_f(2);
f(5);
```

(global) **I**
x: 10
make_f: function... vs=**I**
f: function... vs=**II**

(make_f) **II**
x:10
a: 2
r: function... vs=**II**

(f) **III**
x:10
a: 2
b: 5
s: 52

Benefits of closures

- Parameterized functions:

```
output = filter(input, function -> r(a)
{ r = less(a, max) })
```

- Generate functions within functions:

```
function make_f -> f(a)
{
  f = function -> y (x) { y = add(x, a)
}
}
```


Higher-order function

- A higher-order function takes t least one of the following
  - takes one or more function as argument
  - returns a function as its result
- All other functions are first-order

Example of higher order function

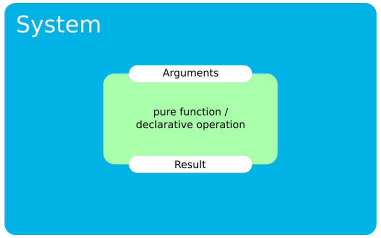
```
function f(a) -> x {
  x = function(b) -> y { y = add(a, b); }
}
map([1, 2, 3],
    function(x) -> y {
      y = add(x, 1);
    })
```

Pure function (1/3)

- A pure function is a function such as:
  - No side effects
  - Always return the same value given the same arguments
- Examples of pure functions:
  - abs, pow, add...
- Examples of non pure function
  - `std.print('Hello World')`
  - `math.rand()`
  - All IO functions

Pure function (2/3)

- They are called *declarative operations* in the book, and formally defined as
  - Independent: depends only on its arguments, nothing else
  - Stateless: no internal state is remembered between calls
  - Deterministic: call with same operations always give same results



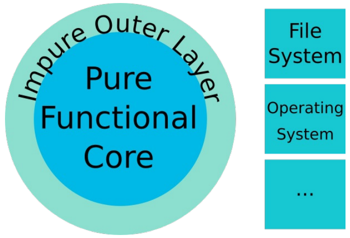
- They can be composed to form other declarative operations

Pure function (3/3)

- To check if a function is pure:
  - Simply check if it only calls pure functions
  - Remember, in functional programming once bounded the value of a variable cannot be changed

Why pure function matters?

- The function definitions tells you all about the behavior
  - Specify exactly what is going in and out



- Make testing/debugging easier
- Reusability, multithreading...

Conclusion

- Declarative Computation Model: Kernel Syntax and Semantic
- Subsitution model: applicative and normal order
- Single assisgnment Store
- Closure
- Pure functions