

Evolution of an interpreter

Ralf Lämmel
Software Language Engineer
University of Koblenz-Landau
Germany

Interpretation
is **the** functional programming domain!

An initial interpreter

```
> interpret (Succ (Succ Zero))  
2
```



We interpret the language of
Peano-like natural numbers.

Elements of an interpreter

- **Syntactic domains**
- Semantic domains
- Semantic functions
- Auxiliary functions

```
data Expr  
  = Zero  
  | Succ Expr
```

Elements of an interpreter

- Syntactic domains
- **Semantic domains**
- Semantic functions
- Auxiliary functions

This is the designated domain for the types of values that interpretation may yield.

```
type Value = Nat  
type Nat = Int
```

We use the non-negative part of Haskell's Int as natural numbers.

Elements of an interpreter

- Syntactic domains
- Semantic domains
- **Semantic functions**
- Auxiliary functions

`interpret :: Expr -> Value`

`interpret Zero = 0`

`interpret (Succ x) = (interpret x) + 1`

For the record:
we use “naive denotational style”.

Elements of an interpreter

- Syntactic domains
- Semantic domains
- Semantic functions
- **Auxiliary functions**

(\$), (+), 0

What's up today?

- The basic art of interpretation
 - Partiality of interpretation
 - Interpretation with type tests
 - Modeling bindings
 - Modeling recursion
 - The Compositionality Principle
- Preparation of lectures to come
 - Monadic style
 - Generalized folds
 - Functional OO Programming

Interpretation
is **the** functional programming domain!

Partiality of interpretation

Return “Just” something
or “Nothing”.

`interpret :: Expr -> Maybe Value`

`> interpret (Succ Zero)`

`Just 1`

`> interpret (Pred Zero)`

`Nothing`

The predecessor extension

- **Syntactic domains**
- ~~Semantic domains~~
- Semantic functions
- Auxiliary functions

```
data Expr
= ...
| Pred Expr
```

The predecessor extension

- Syntactic domains
- ~~Semantic domains~~
- **Semantic functions**
- Auxiliary functions

```
interpret Zero
  = Just 0
```

```
interpret (Succ x)
  = (Just . (+1)) $$ interpret x
```

```
interpret (Pred x)
  = pred $$ interpret x
where
```

```
pred n | n > 0      = Just (n-1)
      | otherwise = Nothing
```

\$ is regular
function application.
\$\$ is partial function
application.

The predecessor extension

- Syntactic domains
- ~~Semantic domains~~
- Semantic functions
- **Auxiliary functions**

`infixr 0 $$`
`($$) :: (a -> Maybe b) -> (Maybe a -> Maybe b)`
`($$) = maybe Nothing`

Function to apply

Argument

`maybe :: b -> (a -> b) -> Maybe a -> b`
`maybe n f Nothing = n`
`maybe n f (Just x) = f x`

[Prelude]

Compositionality

Each term (aka phrase) is given a meaning that describes its contribution to the meaning of a complete term (say, program) that contains it. More technically, **the meaning of each term is given as a function of the meanings of its immediate sub-terms.**

`interpret Zero = fZero`

`interpret (Succ x) = fSucc (interpret x)`

`interpret (Pred x) = fPred (interpret x)`

Those f_i do not
refer to syntax!

Who cares about compositionality?

- (Mathematicians, logicians, linguists, ...)
- Language engineers
 - Prove properties of constructs.
 - Prove correctness of transformation.
 - ...

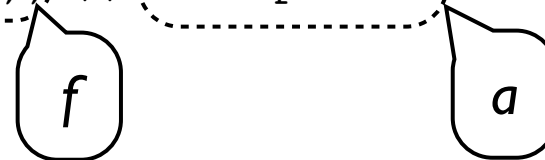
Example (lemma): interpretation of the sub-language **Zero** + **Succ** never fails.

- Proof by **structural induction**
 - *Induction hypothesis (IH)*
Interpretation of subterms never fails.
 - *Base case*
Interpretation of **Zero** does not fail.
 - *Inductive step for case Succ*
Meaning is of form $f \$\$ a$ where
 - a does not fail per IH, and
 - f does not fail due to the form **Just** ...

```
interpret Zero = Just 0
```

```
interpret (Succ x) = ((Just . (+1))) $$ interpret x
```

```
interpret (Pred x) = ...
```



How would we destroy compositionality?

- Use side effects.
- Use deep patterns, e.g.:

```
interpret (Pred (Succ x))  
  
= interpret x
```

- Define meaning in terms of syntax, e.g.:

```
interpret (While c b)  
  
= interpret (If c (Seq b (While c b)))
```

Riddle: modularize the current interpreter to work out three modules: *Syntax*, *Meanings*, *Interpreter*. The *Meanings* module does not import the *Syntax* module.

```
interpret Zero = fZero  
interpret (Succ x) = fSucc (interpret x)  
interpret (Pred x) = fPred (interpret x)
```


Interpretation with type tests

Consider the following demo
based on a simple extension
for Boolean constructs:

```
> let n0 = Zero  
> let n1 = Succ n0  
> interpret (Cond (IsZero n0) n1 n0)  
Just 1
```

The extension for conditionals

- **Syntactic domains**
- Semantic domains
- Semantic functions
- Auxiliary functions

```
data Expr = ...  
    | IsZero Expr  
    | Cond Expr Expr Expr
```

One might think that distinct syntactic categories for Boolean vs. number expressions would make type tests unnecessary, but this approach does not scale: think of variables.

The extension for conditionals

- Syntactic domains
- **Semantic domains**
- Semantic functions
- Auxiliary functions

```
data Value
  = InNat Nat
  | InBool Bool
```

The extension for conditionals

- Syntactic domains
- Semantic domains
- **Semantic functions**
- Auxiliary functions

```
interpret :: Expr -> Maybe Value
interpret Zero          = zero
interpret (Succ x)      = succ $$ interpret x
interpret (Pred x)      = pred $$ interpret x
interpret (IsZero x)    = isZero $$ interpret x
interpret (Cond xc xt xe) = cond yc yt ye
  where
    yc = interpret xc
    yt = interpret xt
    ye = interpret xe
```



“Obviously”
compositional style

Auxiliary functions for composing meanings:
we need injections and projections everywhere.

```
zero :: Maybe Value
zero = Just (InNat 0)
```

```
succ :: Value -> Maybe Value
succ = ($$) (Just . InNat . (+1)) . outNat
```

```
pred :: Value -> Maybe Value
pred = ...
```

```
isZero :: Value -> Maybe Value
isZero = ($$) (Just . InBool . (==0)) . outNat
```

```
cond rc rt re = cond' $$ (outBool) $$ rc)
  where
    cond' b = if b then rt else re
```

The extension for conditionals

- Syntactic domains
- Semantic domains
- Semantic functions
- **Auxiliary functions**

Partial projections
as opposed to total
injections.

```
outNat :: Value -> Maybe Nat
outNat (InNat n) = Just n
outNat _ = Nothing
```

```
outBool :: Value -> Maybe Bool
outBool (InBool b) = Just b
outBool _ = Nothing
```

Modeling bindings

Consider the following demo
based on a simple extension
for the lambda calculus:

```
> let inc = Lambda "x" (Succ (Var "x"))  
> interpret (Apply inc Zero) (const Nothing)  
Just 1
```

The lambda calculus

- **Syntactic domains**
- Semantic domains
- Semantic functions
- Auxiliary functions

```
data Expr = ...  
  | Var String  
  | Lambda String Expr  
  | Apply Expr Expr
```


The lambda calculus

- Syntactic domains
- **Semantic domains**
- Semantic functions
- Auxiliary functions

Functions in addition to natural numbers and Booleans

```
data Value = ...  
  | InFun (Value -> Maybe Value)
```

```
type Env = String -> Maybe Value
```

Environments to hold on the bindings of lambda variables

The lambda calculus

- Syntactic domains
- Semantic domains
- **Semantic functions**
- **Auxiliary functions**

Lift previous equations to new semantic model, and add equations for new constructs.

Full rewrite of the previous semantic function

```

interpret :: Expr -> Env -> Maybe Value
interpret Zero _ = zero
interpret (Succ x) e = succ $$ interpret x e
interpret (Pred x) e = pred $$ interpret x e
interpret (IsZero x) e = isZero $$ interpret x e
interpret (Cond xc xt xe) e = cond yc yt ye
  where
    yc = interpret xc e
    yt = interpret xt e
    ye = interpret xe e

```

New construct **Var**

```
interpret (Var n) e = var e n
```

```
var :: Env -> String -> Maybe Value  
var = ($) 
```



Environment
lookup is function
application!

```
interpret (Lambda n x) e
  = lambda e n (interpret x)
```

```
lambda :: Env
        -> String
        -> (Env -> Maybe Value)
        -> Maybe Value
```

```
lambda e n f
  = Just (InFun (\r ->
                  f (modify e n (Just r))))
```

```
modify :: Eq x
        => (x -> y)
        -> x -> y -> x -> y
modify f x y x'
  = if x==x' then y else f x'
```

New construct Lambda

New construct **Apply**

```
interpret (Apply xf xa) e
  = apply (interpret xf e) (interpret xa e)
```

```
apply :: Maybe Value
      -> Maybe Value
      -> Maybe Value
apply f a
  = (\f' ->
     (\a' -> (flip ($) a')
              $$ outFun f')
     $$ a)
     $$ f
```

Riddle: derive a variation of the interpreter so that possibly failing or diverging function arguments are evaluated more lazily.

Modeling recursion

```
> let fac = ???  
> let s = Succ  
> let x5 = s (s (s (s (s Zero))))  
> let e = const Nothing  
> interpret (Apply fac x5) e  
Just 120
```


The least fixed point combinator of Haskell

```
> let fac f x = if x==0 then 1 else x * f (x-1)
```

```
> let fix f = f (fix f)
```

```
> fix fac 5
```

```
120
```

Interpreted factorial function

```
fac :: Expr
```

```
fac
```

```
= Apply fix  
  (Lambda "f"  
    (Lambda "x"  
      (Cond (IsZero (Var "x"))  
            (Succ Zero)  
            (Apply  
              (Apply  
                mult  
                (Var "x"))  
              (Apply  
                (Var "f")  
                (Pred (Var "x"))))))))
```

Better be a lambda
expression!

The call-by-value fixed point combinator of the lambda calculus

```
fix :: Expr
fix = Lambda "f" (Apply t t)
  where
    t = Lambda "x"
      (Apply f
        (Lambda "y"
          (Apply (Apply x x) y)))
    where
      f = Var "f"
      x = Var "x"
      y = Var "y"
```

A language construct for recursive bindings

```
Letrec "add" (... "add" ...)
  (Letrec "mult" (... "mult" ...)
    (Letrec "fac" (... "fac" ...)
      (Apply (Var "fac" ...) )))
```

The **Letrec** construct

```
data Expr
= ...
| Letrec n Expr Expr
```



Wow!

```
interpret (Letrec n x1 x2) e
= interpret x2 (fix ( modify e n
                        . interpret x1))
```

```
fix :: (x -> x) -> x
fix f = f (fix f)
```

Riddle: define a type checker for our applicative lambda calculus with letrec. Basic types are “Nat”, “Bool”, and type variables. Compound types are constructed through (\rightarrow) .

Ideas for more extensions

- Recursive binding groups
- Records and objects
- States as side effects
- Exception handling
- Nondeterminism
- ...

Further reading

- John C. Reynolds:
Definitional Interpreters for Higher-Order Programming Languages
Definitional Interpreters Revisited
- R.D.Tennet:
Semantics of Programming Languages
- Kenneth Slonneger and Barry L. Kurtz:
Syntax and Semantics of Programming Languages
- Shriram Krishnamurthi:
Programming Languages: Application and Interpretation

What's up next?



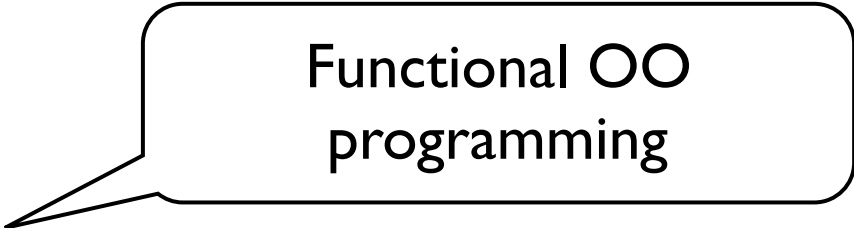
Effects
(monads & transformers)

- Let's abstract from partiality et al.!



Generalized folds
(bananas)

- What's compositional style really?



Functional OO
programming

- How to do all this in C#?

Thanks!
Questions and comments welcome.