Function Totality: Abstraction Tool in Programming

Marko Dimjašević, PhD

Lambda Days, Kraków, Poland, 21 February 2019

INPUT | OUTPUT

About Me

- From Croatia
- Formal Methods Engineer at Input Output Hong Kong
- Doctoral dissertation at University of Utah, USA on automatic software testing
- Research on runtime verification at NASA
- Interested in:
  - Software correctness via type systems
  - Reducing software complexity via embedded domain specific languages

In This Talk

Introduction

Exhaustive Pattern Matching

Termination

Productivity

Totality

Conclusions

Introduction

- Function totality = termination + productivity
  - Key to: 1) reducing the number of runtime errors, and 2) the ability to abstract
  - Exhaustiveness important too
- Types have a central role
- Function definition via equations and pattern matching
- Code examples: Haskell, Idris

Introduction: Abstraction

"The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise." $\sim$ *Edsger W. Dijkstra*

Introduction: Software Components

- Function as a software component
- Use smaller components with known functionality to compose bigger components
- Abstraction over details from smaller components

Introduction: Haskell, Idris

- Pure functional programming languages
  - *Programming like doing mathematics*
- Haskell first appeared in 1990, Idris in 2009
- Idris strictly evaluates by default
- Haskell's type system based on parametric polymorphism
  - Algebraic data types
  - Idris: dependent types
- General-purpose programming languages
  - Idris also a theorem proving assistant

Introduction: The Role of Types

- Type: a set of values

```
data Vehicle = Car | Motorcycle
data Person = MkPerson Int Vehicle
```

- Function: maps a set to another set

```
getVehicle :: Person -> Vehicle
getVehicle (MkPerson age vehicle) = vehicle
```

- Types determine the kind of data that functions work with
- Types direct termination and productivity checking
- Compiler: performs automatic checks if expected and actual types match

Exhaustive Pattern Matching

- Have we covered all cases of input values?
- Fetch the head of a list

```
head        :: [a] -> a
head (x:_) =  x
```

- Inexhaustive pattern matching: no empty list case
  - `head` is a partial function

Exhaustive Pattern Matching: List Head Error

- Cover the whole function domain

```
head        :: [a] -> a
head (x:_) =  x
head []     =  error "empty list"
```

- Runtime error when `head []` called
- Problem: The type of the function is not appropriate

Exhaustive Pattern Matching: Different Codomain

Choose a better codomain

```
head          :: [a] -> Maybe a
head (x:_) =  Just x
head []       =  Nothing
```

Exhaustive Pattern Matching: Vehicle Example

- Example: greet a vehicle owner
  - If underage (18), they should have no vehicle
  - If at least 18, they have a car or a motorcycle
- Three cases in total:
  1. The person is underage and therefore cannot own a vehicle
  2. The person is at least 18 and has a car
  3. The person is at least 18 and has a motorcycle

Exhaustive Pattern Matching: Haskell

```
data Vehicle = Car | Motorcycle
data Person = MkPerson Int Vehicle

limit = 18

greet :: Person -> String
greet (MkPerson age _)          | age <  limit =
  "Be patient, you're not old enough to drive!"
greet (MkPerson age Car)        | age >= limit =
  "Hello, you car driver!"
greet (MkPerson age Motorcycle) | age >= limit =
  "Hello, you motorcycle driver!"
```

Exhaustive Pattern Matching: GHC Warning

```
$ ghc -Wincomplete-patterns Greet.hs
[1 of 1] Compiling Greet            ( Greet.hs, Greet.c

Greet.hs:7:1: warning: [-Wincomplete-patterns]
    Pattern match(es) are non-exhaustive
    In an equation for ‘greet’:
Patterns not matched:
    (MkPerson _ Car)
    (MkPerson _ Motorcycle)
   |
7  | greet (MkPerson age _)          | age <  limit =
   | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^..
```

INPUT | OUTPUT

Exhaustive Pattern Matching: Idris (1)

```
data Vehicle = Car | Motorcycle

possiblyVehicle : Nat → Type
possiblyVehicle n = if n < 18 then () else Vehicle

data Person : Type where
  MkPerson : (age : Nat) → (v : possiblyVehicle age)
Person

p1 : Person
p1 = MkPerson 11 ()

-- It will not type-check
-- p2 : Person
-- p2 = MkPerson 16 Car

p3 : Person
p3 = MkPerson 24 Motorcycle
```

Exhaustive Pattern Matching: Idris (2)

```
data Vehicle = Car | Motorcycle

possiblyVehicle : Nat → Type
possiblyVehicle n = if n < 18 then () else Vehicle

data Person : Type where
  MkPerson : (age : Nat) → (v : possiblyVehicle age) →
Person

greet : Person → String
greet (MkPerson age v) with (age < 18)
  greet (MkPerson _ ())           | True  =
    "Be patient, you're not old enough to drive!"
  greet (MkPerson _ Car)          | False =
    "Hello, you car driver!"
  greet (MkPerson _ Motorcycle) | False =
    "Hello, you motorcycle driver!"
```

Exhaustive Pattern Matching: Idris Check

- The Idris compiler checks for exhaustiveness

```
:total greet
Greet.greet is Total
```

- The `greet` function exhaustively covers all possible shapes and values of type `Person`

Termination

- Will the program eventually finish running given an input?

```
length :: [a] -> Word
length []       = 0
length (x : xs) = 1 + length xs
```

Termination: Loop

- Will this program terminate? (taken from the paper Total Functional Programming)

```
loop :: Int -> Int
loop n = 1 + loop n
```

Termination: Mathematical Reasoning

- Mathematical reasoning in functional programming

  ```
  loop :: Int -> Int
  loop n = 1 + loop n
  ```

- Substitute `0` for `n`:

  ```
  loop 0 = 1 + loop 0
  ```

- Assume `x - x = 0` and subtract `loop 0` from both sides to get:

  ```
  0 = 1
  ```

- What went wrong?

Termination: Bottom Value

- We went from the program

  ```
  loop :: Int -> Int
  loop n = 1 + loop n
  ```

  to

  ```
  0 = 1
  ```

- `n` is not only an integer, but also a bottom (undefined integer)
- An infinite loop in programming corresponds to falsity in logic
  - `loop` is a partial function, hence not suitable for equational reasoning

Termination: Halting Problem

- The Halting Problem in computability theory
  - Given a program description and an input, will the program finish with its execution?
  - In 1936 Alan Turing proved there is no general algorithm that addresses this question
- How can Idris check for termination?
  - Restriction to a function class for which it is doable (adapt the style of program writing)

Termination: Recap

- Inexhaustive pattern matching and infinite loops

```
head        :: [a] -> a
head (x:_) =  x

loop :: Int -> Int
loop n = 1 + loop n
```

- To rely on such functions calls for trouble: dreadful bug searching and fixing
- A terminating function:
    1. Is defined for all well-typed inputs, and
    2. Converges on a base case in the recursive call.

Productivity

- What about programs that should not terminate, e.g., an operating system or a web server?
  - Such programs produce data for a given input and keep on doing that in a loop
- Productivity: giving a non-empty finite prefix of an infinite result in finite time

Productivity: Infinite Looping (1)

- An adapted example from the book Type-driven Development with Idris
- An ever-running process printing to the console
- How to check it is productive?

Productivity: Infinite Looping (2)

```
data InfIO : Type where
  Do : IO a → (a → Inf InfIO) → InfIO

infProg : InfIO
infProg = Do (putStrLn "Lambda") (λ_ ⇒ infProg)

partial
run' : InfIO → IO ()
run' (Do c f) = do res ← c
                   run' (f res)
```

Productivity: Fuel (1)

- Termination checking for indefinitely running programs: fuel consumption as a guaranty of getting to a final state with no fuel
- Infinite fuel tank
  - Pushing infinite execution out of a critical program part

Productivity: Fuel (2)

```
%default total

data Fuel = Dry | More (Lazy Fuel)

twoDrops : Fuel
twoDrops = More (More Dry)

partial
forever : Fuel
forever = More forever
```

Productivity: Fuel (3)

```
data InfIO : Type where
  Do : IO a → (a → Inf InfIO) → InfIO

infProg : InfIO
infProg = Do (putStrLn "Lambda") (λ_ ⇒ infProg)

run : Fuel → InfIO → IO ()
run (More fuel) (Do c f) = do res ← c
                              run fuel (f res)
run Dry         _        = putStrLn "No more fuel"
```

INPUT | OUTPUT

Productivity: Fuel (4)

When executed with two drops of fuel:

```
:exec run twoDrops infProg
Lambda
Lambda
No more fuel
```

When executed with infinite fuel:

```
:exec run forever infProg
Lambda
Lambda
Lambda
Lambda
Lambda
...
```

Productivity: Fuel (5)

The `run` function is total:

```
:total run
RunFuel.run is Total
```

Totality

- Function totality comprises termination and productivity
- A total function:
  1. Terminates its execution for a given well-typed data input, or
  2. Produces a non-empty finite prefix of the result of the correct type in finite time

Totality: Program Parts

- Programs can be split into a finite and an infinite part:
  1. The finite part always has to be total
  2. The infinite part has to be as productive as possible
     - The possibility of runtime error only in the partial part of the infinite part

Totality: Recap

- Totality: termination and productivity
- Safe mathematical reasoning about total functions
- A link to the Curry-Howard isomorphism
    - If I had a partial proof, how would I reuse it in more complex proofs?

Literature

- Paper by David Turner: Total Functional Programming
- Aaron Stump: Verified Functional Programming in Agda, chapter 9 (termination proofs)
- Edwin Brady: Type-driven Development with Idris
- Daniel Friedman, David Christiansen: The Little Typer

Conclusions

- Composing smaller functions into bigger functions
- Totality: terminating and productive functions
  - Supports abstraction
- Define functions over the whole domain:
  - Exhaustive pattern matching
  - Fix the domain or the codomain
- Compiler as a verification tool