Say it intensionally

Jean-François MONIN

More than two decades ago

Olivier Danvy : *Functional unparsing*

Journal of Functional Programming, 8(6): 621–625 1998

"Unparsing" just means `sprintf`

A dirty-looking function provided in the C language and in OCaml

More than two decades ago

Olivier Danvy : *Functional unparsing*

Journal of Functional Programming, 8(6): 621–625 1998

"Unparsing" just means `sprintf`

A dirty-looking function provided in the C language and in OCaml

More than two decades ago

Olivier Danvy : *Functional unparsing*
Journal of Functional Programming, 8(6): 621–625 1998

“Unparsing” just means `sprintf`

A dirty-looking function provided in the C language and in OCaml

sprintf

The following code

```
sprintf "The %s is %i %s." "distance" 10 "meters"
```

returns the string

```
"The distance is 10 meters."
```

The first argument is called the *format*.
The number of following arguments and their types depend on the format.

sprintf

The following code

```
sprintf "The %s is %i %s." "distance" 10 "meters"
```

returns the string

```
"The distance is 10 meters."
```

The first argument is called the *format*.
The number of following arguments and their types depend on the format.

Functional sprintf

```
"The %s is %i %s."
```

With dependent types

The format is implemented by a *list* of directives, using a suitable sum type of directives.

```
[Lit "The "; String; Lit " is "; Int; String; Lit "."]
```

Danvy' trick

The format is implemented by a *function*

```
lit "The " ∘ str  ∘ lit " is "  ∘ sint  ∘ str  ∘ lit "."
lit := fun s k a -> k (a ++ s)
str := fun k a s -> k (a ++ s)

sprintfk := fun f -> f id ""
```

Functional sprintf

```
"The %s is %i %s."
```

With dependent types

The format is implemented by a *list* of directives, using a suitable sum type of directives.

```
[Lit "The "; String; Lit " is "; Int; String; Lit "."]
```

Danvy' trick

The format is implemented by a *function*

```
lit "The " ∘ str  ∘ lit " is "  ∘ sint  ∘ str  ∘ lit "."
lit := fun s k a -> k (a ++ s)
str := fun k a s -> k (a ++ s)


sprintfk := fun f -> f id ""
```

Functional sprintf

```
"The %s is %i %s."
```

With dependent types

The format is implemented by a *list* of directives, using a suitable sum type of directives.

```
[Lit "The "; String; Lit " is "; Int; String; Lit "."]
```

Danvy' trick

The format is implemented by a *function*

```
lit "The " ∘ str  ∘ lit " is "  ∘ sint  ∘ str  ∘ lit "."
lit := fun s k a -> k (a ++ s)
str := fun k a s -> k (a ++ s)

sprintfk := fun f -> f id ""
```

Exercise: prove that `sprintf` does the job (*)

```
r_sprintf (l : list directive) : string -> type_of l
kformat (l : list directive) :
   (string -> string) -> (string -> type_of l)
```

By induction on the format `l`

$\forall a,$ `r_sprintf l a = kformat l id` a

FAILS!

```
   (fun x -> r_sprintf l (a ++ x))
=  (fun x -> kformat l string id (a ++ x))
```

Extensionality?

(*) JFM, TPHOL'2004

Exercise: prove that `sprintf` does the job (*)

```
r_sprintf (l : list directive) : string -> type_of l
kformat (l : list directive) :
   (string -> string) -> (string -> type_of l)
```

By induction on the format `l`

$\forall a,$ `r_sprintf l a = kformat l id` a

FAILS!

```
   (fun x -> r_sprintf l (a ++ x))
=  (fun x -> kformat l string id (a ++ x))
```

Extensionality?

(*) JFM, TPHOL'2004

Exercise: prove that `sprintf` does the job (*)

```
r_sprintf (l : list directive) : string -> type_of l
kformat (l : list directive) :
   (string -> string) -> (string -> type_of l)
```

By induction on the format `l`

$\forall a,$ `r_sprintf l a = kformat l id` a

FAILS!

```
   (fun x -> r_sprintf l (a ++ x))
=  (fun x -> kformat l string id (a ++ x))
```

Extensionality?

(*) JFM, TPHOL'2004

What is a function (from A to B)?

In Set Theory

- A special relation subset of $A \times B$ with unicity of output
- A set f of pairs (a, b)
  such that $\forall a\, b_1\, b_2,\ (a, b_1) \in f$ and $(a, b_2) \in f$ implies $b_1 = b_2$.

What is $b_1 = b_2$?

- What is equality ?
- A relation?
- A function??

What is a function (from A to B)?

In Set Theory

- A special relation subset of $A \times B$ with unicity of output
- A set f of pairs (a, b)
  such that $\forall a\, b_1\, b_2,\ (a, b_1) \in f$ and $(a, b_2) \in f$ implies $b_1 = b_2$.

What is $b_1 = b_2$?

- What is equality ?
- A relation?
- A function??

What is a function (from A to B)?

In Set Theory

- A special relation subset of $A \times B$ with unicity of output
- A set f of pairs (a, b)
  such that $\forall a\, b_1\, b_2,\ (a, b_1) \in f$ and $(a, b_2) \in f$ implies $b_1 = b_2$.

What is $b_1 = b_2$?

- What is equality ?
- A relation?
- A function??

Leibniz principle

- b_1 can be replaced by b_2 everywhere
- In particular, at its first occurrence in the sentence "b_1 can be replaced by b_1 everywhere"
- We get symmetry

Leibniz principle

- b_1 can be replaced by b_2 everywhere
- In particular, at its first occurrence in the sentence
  "b_1 can be replaced by b_1 everywhere"
- We get symmetry

What is $f_1 = f_2$?

Where f_1 and f_2 : functions from A to B

In Set Theory

f_1 and f_2 agree on all inputs
$\forall a\, b_1\, b_2,\ (a, b_1) \in f_1$ and $(a, b_2) \in f_2$ implies $b_1 = b_2$.

In practice

Impossible to check (in a brutal way) if A is `nat`
Or worse: if A contains functions from `nat` to `nat`, etc.

What is $f_1 = f_2$?

Where f_1 and f_2 : functions from A to B

In Set Theory

f_1 and f_2 agree on all inputs
$\forall a\, b_1\, b_2,\ (a, b_1) \in f_1$ and $(a, b_2) \in f_2$ implies $b_1 = b_2$.

In practice

Impossible to check (in a brutal way) if A is `nat`
Or worse: if A contains functions from `nat` to `nat`, etc.

What is $f_1 = f_2$?

Where f_1 and f_2 : functions from A to B

In Set Theory

f_1 and f_2 agree on all inputs
$\forall a\, b_1\, b_2,\ (a, b_1) \in f_1$ and $(a, b_2) \in f_2$ implies $b_1 = b_2$.

In practice

Impossible to check (in a brutal way) if A is `nat`
Or worse: if A contains functions from `nat` to `nat`, etc.

In programming, e.g. lambda-calculus

- Let
  - f be some function
  - usual composition $\circ$ defined by $g \circ f := \lambda x.\, g(f\,x)$
- Consider $f_1 := f \circ (f \circ f)$ and $f_2 := (f \circ f) \circ f$
- By reduction $f_1 := \lambda x.\, f(f\,(f\,x))$ and $f_2 := \lambda x.\, f(f\,(f\,x))$

Theorem 1

f_1 and f_2 are the same program

Corollary 2

f_1 and f_2 are extensionally equal

In programming, e.g. lambda-calculus

- Let
  - f be some function
  - usual composition $\circ$ defined by $g \circ f := \lambda x.\, g(f\,x)$
- Consider $f_1 := f \circ (f \circ f)$ and $f_2 := (f \circ f) \circ f$
- By reduction $f_1 := \lambda x.\, f(f\,(f\,x))$ and $f_2 := \lambda x.\, f(f\,(f\,x))$

Theorem 1

f_1 and f_2 are the same program

Corollary 2

f_1 and f_2 are extensionally equal

In programming, e.g. lambda-calculus

- Let
  - f be some function
  - usual composition $\circ$ defined by $g \circ f \ := \ \lambda x.\, g\,(f\,x)$
- Consider $f_1 := f \circ (f \circ f)$ and $f_2 := (f \circ f) \circ f$
- By reduction $f_1 := \lambda x.\, f(f\,(f\,x))$ and $f_2 := \lambda x.\, f(f\,(f\,x))$

Theorem 1

f_1 and f_2 are the same program

Corollary 2

f_1 and f_2 are extensionally equal

In programming, e.g. lambda-calculus

- Let
  - f be some function
  - usual composition $\circ$ defined by $g \circ f \; := \; \lambda x.\, g\,(f\,x)$
- Consider $f_1 := f \circ (f \circ f)$ and $f_2 := (f \circ f) \circ f$
- By reduction $f_1 := \lambda x.\, f(f\,(f\,x))$ and $f_2 := \lambda x.\, f(f\,(f\,x))$

Theorem 1

f_1 and f_2 are the same program

Corollary 2

f_1 and f_2 are extensionally equal

(Leibniz) equality between (functional) programs

Finitist answers provided by computer science

without functional extensionality

- Same code
- Same code up to preliminary reductions
  (static execution at compile time)

(Leibniz) equality between (functional) programs

Finitist answers provided by computer science

without functional extensionality

- Same code
- Same code up to preliminary reductions
  (static execution at compile time)

(Leibniz) equality between (functional) programs

Finitist answers provided by computer science

without functional extensionality

- Same code
- Same code up to preliminary reductions (static execution at compile time)

Exercise: prove that `sprintfk` does the job

By induction on the format `l`

$\forall a,$ `r_sprintf l a = kformat l id` a

FAILS

By very short induction on the format `l`

`r_sprintf l = kformat l id`

WORKS :)

Key hint:

```
    fun a x -> r_sprintf l (a ++ x)
 =  some_suitable_higher_order_function (r_sprintf l)
```

Exercise: prove that `sprintfk` does the job

By induction on the format `l`

$\forall a,$ `r_sprintf l a = kformat l id` a

FAILS

By very short induction on the format `l`

`r_sprintf l = kformat l id`

WORKS :)

Key hint:

```
    fun a x -> r_sprintf l (a ++ x)
 =  some_suitable_higher_order_function (r_sprintf l)
```

Exercise: prove that `sprintfk` does the job

By induction on the format `l`

$$\forall a, \texttt{r\_sprintf l a = kformat l id } a$$

FAILS

By very short induction on the format `l`

```
r_sprintf l = kformat l id
```

WORKS :)

Key hint:

```
    fun a x -> r_sprintf l (a ++ x)
 =  some_suitable_higher_order_function (r_sprintf l)
```

Many simpler examples on ordinary lists

Easier to understand with the following version of app, called φ

```
φ u          := id
φ (x :: u)   := (cons x) ∘ (φ u)
```

φ is a (computational) morphism

```
φ (u ++ v) ==  φ u ∘ φ v
```

```
rev u          := id
rev (x :: u)   :=  (rev u) ∘ (cons x)
```

This definition makes no use of app, and is linear time complexity

Receipt

- write the simple minded definition with app
- replace app by $\circ$