Diffeological spaces as a model for differentiable programs

A tutorial

Philip Saville, University of Oxford

these slides available at philipsaville.co.uk

(1) What questions does denotational semantics study?
(2) Why are cartesian closed categories so important?
(3) Where do diffeological spaces come in?

What does programming language theory study?

We want programs that are:

efficient, fast, and correct

We ask:

(1) When are programs interchangeable?
(2) How should we think about programs?

gets interesting when programs have effects
= interaction with the world

When are programs interchangeable?

example 1

flips a coin with bias $p \in [0,1]$

```
let b = flip(p);
return b;
```

$\xrightarrow[\text{not correct!}]{\text{more efficient}}$

```
return (heads)
```

```
let b = flip(p);
if b == heads:
   then return (heads);
   else return (heads);
```

$\xrightarrow[\text{correct!}]{\text{more efficient}}$

```
return (heads)
```

When are programs interchangeable?

example 2

fun $\text{double}_1(n)$:
  set_memory $\text{location}_1 := n$;
  return (
    get_memory (location_1)
      + get_memory (location_1)
  );

fun $\text{double}_2(n)$:
  set_memory $\text{location}_1 := n$
  set_memory $\text{location}_2 := n$
  return (
    get_memory (location_1)
      + get_memory (location_2)
  );

$\text{double}_1(n)$

memory		memory
$\text{location}_1 := 0$	→	$\text{location}_1 := n$
$\text{location}_2 := 0$		$\text{location}_2 := 0$
…		…
$\text{location}_k := 0$		$\text{location}_k := 0$

$\text{double}_2(n)$

memory		memory
$\text{location}_1 := 0$	→	$\text{location}_1 := n$
$\text{location}_2 := 0$		$\text{location}_2 := n$
…		…
$\text{location}_k := 0$		$\text{location}_k := 0$

When are programs interchangeable?

example 2

$\mathrm{double}_1(n)$

memory		memory
$location_1 := 0$		$location_1 := n$
$location_2 := 0$	$\longrightarrow$	$location_2 := 0$
…		…
$location_k := 0$		$location_k := 0$

$\mathrm{double}_2(n)$

memory		memory
$location_1 := 0$		$location_1 := n$
$location_2 := 0$	$\longrightarrow$	$location_2 := n$
…		…
$location_k := 0$		$location_k := 0$

equal as functions but not as programs!

$\rightsquigarrow$ we can observe a difference in behaviour

When are programs interchangeable?

example 2

$\mathrm{double}_1(n)$

memory

location$_1$:= 0
location$_2$:= 0
...
location$_k$:= 0

$\longrightarrow$

memory

location$_1$:= n
location$_2$:= 0
...
location$_k$:= 0

$\mathrm{double}_2(n)$

memory

location$_1$:= 0
location$_2$:= 0
...
location$_k$:= 0

$\longrightarrow$

memory

location$_1$:= n
location$_2$:= n
...
location$_k$:= 0

equal as functions but not as programs!

$\rightsquigarrow$ we can observe a difference in behaviour

$$\forall n \,.\, \mathrm{double}_1(n) = \mathrm{double}_2(n)$$

When are programs interchangeable?

example 2

$double_1(n)$

memory		memory
location$_1$:= 0 location$_2$:= 0 … location$_k$:= 0	$\longrightarrow$	location$_1$:= n location$_2$:= 0 … location$_k$:= 0

$double_2(n)$

memory		memory
location$_1$:= 0 location$_2$:= 0 … location$_k$:= 0	$\longrightarrow$	location$_1$:= n location$_2$:= n … location$_k$:= 0

equal as functions but not as programs!

$\rightsquigarrow$ we can observe a difference in behaviour

$$\forall n \,.\, double_1(n) = double_2(n)$$

but can distinguish them by looking at memory:

```
double_i(2);
let n = get_memory location_2;
if n > 0:
   then  return (false);
   else return (true);
```

i = 1 $\swarrow$ return (true);

i = 2 $\searrow$ return (false);

When are programs interchangeable?

programs P and Q are observationally equivalent
if there's no way to observe a difference in behaviour

any program $\mathscr{C}[P]$ containing P gives a result
iff $\mathscr{C}[Q]$ gives the same result

What does programming language theory study?

We want programs that are:

efficient, fast, and correct

We ask:

(1) When are programs interchangeable?

(2) How should we think about programs?

What does programming language theory study?

We want programs that are:

efficient, fast, and correct

We ask:

(1) When are programs interchangeable?

(2) How should we think about programs?

observational equivalence

generally harder than function equality

depends on the language's features

depends on how programs run

Observational equivalence in the real world

how do you prove you're not Banksy?

A town councillor has resigned, blaming people who falsely accused him of being the world famous artist Banksy.

Pembroke Dock councillor William Gannon said the "quite ridiculous" claims were made on several social media pages.

In his resignation letter he claimed this was "undermining my ability to do the work" of a councillor.

Mr Gannon has since made an "I am not Banksy" badge to avoid any confusion and said he would now be returning to his former role of community artist.

He said the allegations meant people were "asking me to prove who I am not and that's almost impossible to do".

https://www.bbc.co.uk/news/uk-wales-61552865

Observational equivalence in the real world

how do you prove you're not Banksy?

A town councillor has resigned, blaming people who falsely accused him of being the world famous artist Banksy.

Pembroke Dock councillor William Gannon said the "quite ridiculous" claims were made on several social media pages.

In his resignation letter he claimed this was "undermining my ability to do the work" of a councillor.

Mr Gannon has since made an "I am not Banksy" badge to avoid any confusion and said he would now be returning to his former role of community artist.

He said the allegations meant people were "asking me to prove who I am not and that's almost impossible to do".

https://www.bbc.co.uk/news/uk-wales-61552865

Observational equivalence in the real world

how do you prove you’re not Banksy?

A town councillor has resigned, blaming people who falsely accused him of being the world famous artist Banksy.

Pembroke Dock councillor William Gannon said the "quite ridiculous" claims were made on several social media pages.

In his resignation letter he claimed this was "undermining my ability to do the work" of a councillor.

Mr Gannon has since made an "I am not Banksy" badge to avoid any confusion and said he would now be returning to his former role of community artist.

He said the allegations meant people were "asking me to prove who I am not and that's almost impossible to do".

https://www.bbc.co.uk/news/uk-wales-61552865

if there’s no way to tell them apart, they must be the same!

What does programming language theory study?

We want programs that are:

efficient, fast, and correct

We ask:

(1) When are programs interchangeable?

(2) How should we think about programs?

observational equivalence

generally harder than function equality

depends on the language's features

depends on how programs run

How should we think about programs?

```
fun add(x, y):
  return (x + y)
```

a function $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$

How should we think about programs?

```
fun add(x, y):
  return (x + y)

fun divide(x, y):
  return (x / y)
```

a function $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$

a function $\mathbb{Z} \times \mathbb{Z}_{\neq 0} \to \mathbb{Q}$

a function $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Q} + \{\mathrm{fail}\}$

How should we **think about programs**?

```
fun add(x, y):
   return (x + y)
```

a function $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$

```
fun divide(x, y):
   return (x / y)
```

a function $\mathbb{Z} \times \mathbb{Z}_{\neq 0} \to \mathbb{Q}$

a function $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Q} + \{\text{fail}\}$

```
fun print_and_return(x):
  print "hello";
  return x;
```

a function $\mathbb{N} \to \{a, b, \ldots, z\}^* \times \mathbb{N}$

$x \mapsto (\text{hello}, x)$

How should we think about programs?

```
fun add(x, y):
   return (x + y)
```

a function $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$

```
fun divide(x, y):
   return (x / y)
```

a function $\mathbb{Z} \times \mathbb{Z}_{\neq 0} \to \mathbb{Q}$

a function $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Q} + \{\text{fail}\}$

```
fun print_and_return(x):
   print "hello";
   return x;
```

a function $\mathbb{N} \to \{a, b, \ldots, z\}^* \times \mathbb{N}$

$x \mapsto (\text{hello}, x)$

```
let b = flip(p);
return b;
```

a probability distribution on $\{\text{heads}, \text{tails}\}$

How should we think about programs?

```
fun add(x, y):
   return (x + y)
```

a function $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$

```
fun divide(x, y):
   return (x / y)
```

a function $\mathbb{Z} \times \mathbb{Z}_{\neq 0} \to \mathbb{Q}$

a function $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Q} + \{\mathrm{fail}\}$

```
fun print_and_return(x):
   print “hello”;
   return x;
```

a function $\mathbb{N} \to \{\mathrm{a}, \mathrm{b}, \ldots, \mathrm{z}\}^* \times \mathbb{N}$

$x \mapsto (\mathrm{hello}, x)$

```
let b = flip(p);
return b;
```

a probability distribution on $\{\mathrm{heads}, \mathrm{tails}\}$

```
normalise(
   let x = sample(bernoulli(0.8));
   let r = (if x then 10 else 3);
   observe 0.45 from exponential(r)
   return(x)
)
```

some measurable function (??)

What does programming language theory study?

We want programs that are:

efficient, fast, and correct

We ask:

(1) When are programs interchangeable?

(2) How should we think about programs?

observational equivalence

generally harder than function equality

depends on the language's features

depends on how programs run

some kind of function?

need something beyond set-theoretic functions to model richer features!

uses ideas from:
- topology
- logic
- order theory

What does programming language theory study?

We want programs that are:

efficient, fast, and correct

We ask:

(1) When are programs interchangeable?
(2) How should we think about programs?

The denotational semantics perspective:

(1) Assign every program P a meaning $[\![P]\!]$
(2) Reason about equality of programs via their meaning
(3) The semantic model tells you what programs ‘really are’

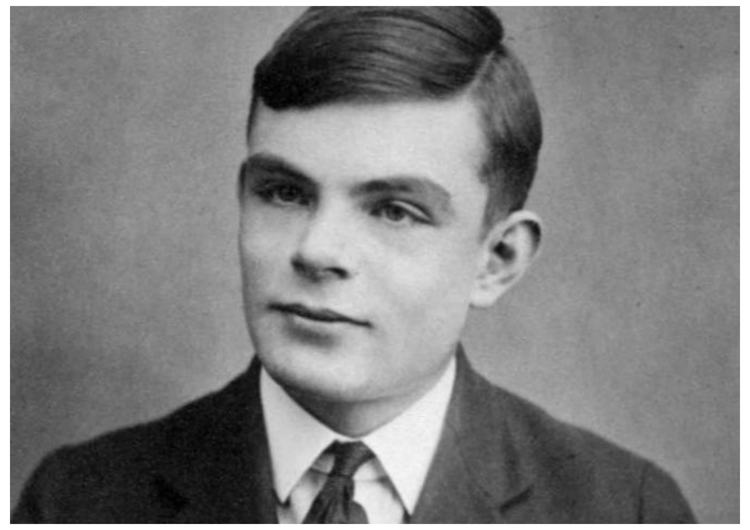
Coming up next

1. Introduce an idealised functional programming language
2. Explain its semantic interpretation in CCCs

---

3. Introduce differentiable programming
4. Explain the interpretation in Diff

What is a program?

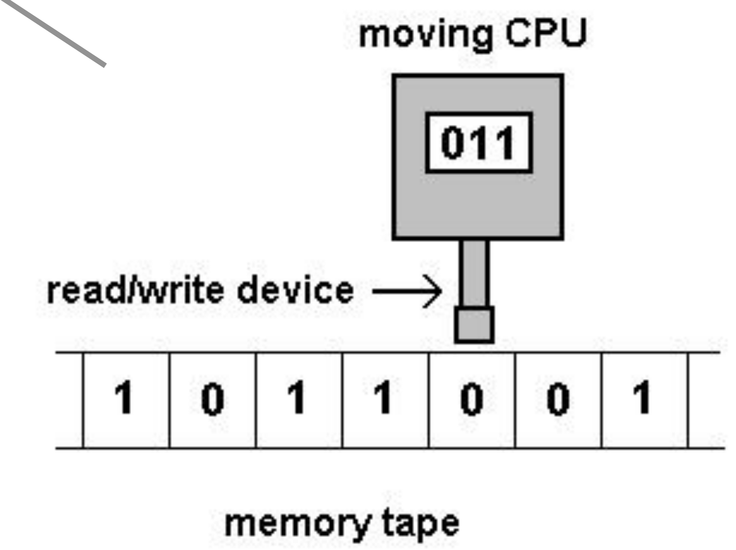


something modelled by a Turing machine

memory you can read
to & write from

Java, C, C++, …

```
n1 = 0
n2 = 1
steps_taken = 0
while (steps_taken < 100) {
    fib = n1 + n2
    n1 = n2
    n2 = fib
    steps_taken = steps_taken + 1
}
```



a kind of function

OCaml, Haskell,
Standard ML,…

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

So a functional programming language lets you
- form functions
- evaluate functions at arguments

How do we define functions?

function body

may not use x, eg $f(x) = 3$

may contain free variables, eg
$f(x) = 3y + x$

$$f(x) = x^3 + x^2 + 1$$

bound variable

the x matters: if

$g(y) = 3y^3 + y^2 + 1$

$h(y) = 3x^3 + x^2 + 1$

then $f = g$ but h is a constant function

every other variable is free

A functional programming language lets you

- form functions
- evaluate functions at arguments

How do we define functions?

A functional programming language lets you

- form functions
- evaluate functions at arguments

function body

may not use x, eg $f(x) = 3$

may contain free variables, eg $f(x) = 3y + x$

$$f(x) = x^3 + x^2 + 1$$

bound variable

the x matters: if

$$g(y) = 3y^3 + y^2 + 1$$

$$h(y) = 3x^3 + x^2 + 1$$

then $f = g$ but h is a constant function

every other variable is free

evaluating = substituting for bound variable

$$f(3) = (x^3 + x^2 + 1)[x \mapsto 3]$$

$$= 3^3 + 3^2 + 1$$

How do we define functions?

function body

may not use x, eg $f(x) = 3$

may contain free variables, eg $f(x) = 3y + x$

A functional programming language lets you

- form functions
- evaluate functions at arguments

$$f(x) = x^3 + x^2 + 1$$ in $\mathbb{R}$ whenever $x \in \mathbb{R}$

bound variable $x \in \mathbb{R}$

the x matters: if

$g(y) = 3y^3 + y^2 + 1$

$h(y) = 3x^3 + x^2 + 1$

then $f = g$ but h is a constant function

every other variable is free

evaluating = substituting for bound variable

$$f(3) = (x^3 + x^2 + 1)[x \mapsto 3]$$

$3 \in \mathbb{R}$

$$= 3^3 + 3^2 + 1$$

How do we define functions?

A functional programming language lets you

- form functions
- evaluate functions at arguments

function body

$$\frac{x^3 + x^2 + 1 \text{ is a program}}{(x \mapsto x^3 + x^2 + 1) \text{ is a program}}$$

bound variable

every other variable is free

$$\frac{(x \mapsto x^3 + x^2 + 1) \text{ is a program} \qquad 3 \text{ is a program}}{(x \mapsto x^3 + x^2 + 1)(3) \text{ is a program}}$$

evaluating = substituting for bound variable

$$(x \mapsto x^3 + x^2 + 1)(3) \rightsquigarrow 3^3 + 3^2 + 1$$

extensionality: $f = \big(x \mapsto f(x)\big)$

$$\big(x \mapsto (x^3 + x^2 + 1)(x)\big) \rightsquigarrow (x \mapsto x^3 + x^2 + 1)$$

How do we define functions?

A functional programming language lets you

- form functions
- evaluate functions at arguments

the λ-calculus

$\lambda x \,.\, f(x) = \big(x \mapsto f(x)\big)$

function body

$$\frac{x^3 + x^2 + 1 \text{ is a program}}{\lambda x \,.\, x^3 + x^2 + 1 \text{ is a program}}$$

bound variable

every other variable is free

$$\frac{\lambda x \,.\, x^3 + x^2 + 1 \text{ is a program} \qquad 3 \text{ is a program}}{(\lambda x \,.\, x^3 + x^2 + 1)(3) \text{ is a program}}$$

evaluating = substituting for bound variable

$$(\lambda x \,.\, x^3 + x^2 + 1)(3) \rightsquigarrow 3^3 + 3^2 + 1$$

extensionality: $f = \big(x \mapsto f(x)\big)$

$$\big(x \mapsto (x^3 + x^2 + 1)(x)\big) \rightsquigarrow (x \mapsto x^3 + x^2 + 1)$$

How do we define functions?

the simply-typed λ-calculus

$\lambda x \,.\, f(x) = \big(x \mapsto f(x)\big)$

A functional programming language lets you

- form functions
- evaluate functions at arguments

function body

$$\frac{x^3 + x^2 + 1 \text{ is a program of type } \mathbb{R}}{\lambda x \,.\, x^3 + x^2 + 1 \text{ is a program of type } \mathbb{R} \to \mathbb{R}}$$

bound variable

every other variable is free

$$\frac{\lambda x \,.\, x^3 + x^2 + 1 \text{ is a program of type } \mathbb{R} \to \mathbb{R} \qquad 3 \text{ is a program of type } \mathbb{R}}{(\lambda x \,.\, x^3 + x^2 + 1)(3) \text{ is a program of type } \mathbb{R}}$$

evaluating = substituting for bound variable

$$(\lambda x \,.\, x^3 + x^2 + 1)(3) \rightsquigarrow 3^3 + 3^2 + 1$$

extensionality: $f = \big(x \mapsto f(x)\big)$

$$\big(x \mapsto (x^3 + x^2 + 1)(x)\big) \rightsquigarrow (x \mapsto x^3 + x^2 + 1)$$

How do we define functions?

the simply-typed λ-calculus

$\lambda x \,.\, f(x) = \big(x \mapsto f(x)\big)$

A functional programming language lets you

- form functions
- evaluate functions at arguments

function body

$$\frac{P \text{ is a program of type } B \qquad x \text{ is a variable of type } A}{\lambda x \,.\, P \text{ is a program of type } A \to B}\ \text{abstraction}$$

bound variable

every other variable is free

$$\frac{P \text{ is a program of type } A \to B \qquad Q \text{ is a program of type } A}{P(Q) \text{ is a program of type } B}\ \text{application}$$

evaluating = substituting for bound variable

$$(\lambda x \,.\, P)(Q) \rightsquigarrow_\beta P[x \mapsto Q]$$

extensionality: $f = \big(x \mapsto f(x)\big)$

$$P \rightsquigarrow_\eta \lambda x \,.\, P(x)$$

How do we define functions?

the simply-typed λ-calculus

$\lambda x\,.\,f(x) = \big(x \mapsto f(x)\big)$

A functional programming language lets you
- form functions
- evaluate functions at arguments

function body

$$\frac{P \text{ is a program of type } B \qquad x \text{ is a variable of type } A}{\lambda x\,.\,P \text{ is a program of type } A \to B}\ \text{abstraction}$$

bound variable

every other variable is free

$$\frac{P \text{ is a program of type } A \to B \qquad Q \text{ is a program of type } A}{P(Q) \text{ is a program of type } B}\ \text{application}$$

evaluating = substituting for bound variable

$$(\lambda x\,.\,P)(Q) \rightsquigarrow_\beta P[x \mapsto Q]$$

extensionality: $f = \big(x \mapsto f(x)\big)$

$$P \rightsquigarrow_\eta \lambda x\,.\,P(x)$$

$$\frac{x \text{ is a variable of type } A}{x \text{ is a program of type } A}$$

How do we define functions?

the simply-typed λ-calculus

$\lambda x \,.\, f(x) = \big(x \mapsto f(x)\big)$

$$\frac{x \text{ is a variable}}{x \text{ is a program}}$$

abstraction
$$\frac{P : B \text{ is a program} \qquad x : A \text{ is a variable}}{\lambda x \,.\, P : A \to B \text{ is a program}}$$

$$\frac{P : A \to B \text{ is a program} \qquad Q : A \text{ is a program}}{P(Q) : B \text{ is a program}}$$
application

evaluating = substituting for bound x

$P(Q) \rightsquigarrow_\beta P[x \mapsto Q]$

= running the program

extensionality

$P \rightsquigarrow_\eta \lambda x \,.\, P(x)$

$f = \big(x \mapsto f(x)\big)$

A functional programming language lets you

- form functions
- evaluate functions at arguments

How do we define functions?

the simply-typed λ-calculus

$\lambda x . f(x) = \big(x \mapsto f(x)\big)$

$$\frac{x \text{ is a variable}}{x \text{ is a program}}$$

abstraction

$$\frac{P : B \text{ is a program} \qquad x : A \text{ is a variable}}{\lambda x . P : A \to B \text{ is a program}}$$

$$\frac{P : A \to B \text{ is a program} \qquad Q : A \text{ is a program}}{P(Q) : B \text{ is a program}}$$

application

evaluating = substituting for bound x

$P(Q) \rightsquigarrow_{\beta} P[x \mapsto Q]$

= running the program

extensionality

$P \rightsquigarrow_{\eta} \lambda x . P(x)$

$f = \big(x \mapsto f(x)\big)$

A functional programming language lets you

- form functions
- evaluate functions at arguments

$$\cfrac{\cfrac{\cfrac{f : A \to B \qquad\qquad x : A}{f(x) : B}}{\lambda x . f(x) : A \to B}}{\lambda f . \lambda x . f(x) : (A \to B) \to (A \to B)}$$

$$\begin{aligned} \mathrm{eval} : (A \Rightarrow B) \times A &\to B \\ (f, x) &\mapsto f(x) \end{aligned}$$

via currying $X \to (A \Rightarrow B) \cong (X \times A) \to B$

How do we define functions?

the simply-typed λ-calculus

$\lambda x\,.\,f(x) = \big(x \mapsto f(x)\big)$

$$\frac{x \text{ is a variable}}{x \text{ is a program}}$$

abstraction

$$\frac{P : B \text{ is a program} \qquad x : A \text{ is a variable}}{\lambda x\,.\,P : A \to B \text{ is a program}}$$

$$\frac{P : A \to B \text{ is a program} \qquad Q : A \text{ is a program}}{P(Q) : B \text{ is a program}}$$

application

evaluating = substituting for bound x

$P(Q) \rightsquigarrow_\beta P[x \mapsto Q]$

= running the program

extensionality

$P \rightsquigarrow_\eta \lambda x\,.\,P(x)$

$f = \big(x \mapsto f(x)\big)$

A functional programming language lets you

- form functions
- evaluate functions at arguments

$$\dfrac{\dfrac{\dfrac{f : A \to B \qquad x : A}{f(x) : B}}{\lambda x\,.\,f(x) : A \to B}}{\lambda f\,.\,\lambda x\,.\,f(x) : (A \to B) \to (A \to B)}$$

$\mathrm{eval} : (A \Rightarrow B) \times A \to B$

$(f, x) \mapsto f(x)$

via currying $X \to (A \Rightarrow B) \cong (X \times A) \to B$

$$\dfrac{\dfrac{\dfrac{g : B \to C \qquad \dfrac{f : A \to B \qquad x : A}{f(x) : B}}{g\big(f(x)\big) : C}}{\lambda x\,.\,g(f\,x) : A \to C}}{\dfrac{\lambda f\,.\,\lambda x\,.\,g(f\,x) : (A \to B) \to (A \to C)}{\lambda g\,.\,\lambda f\,.\,\lambda x\,.\,g(f\,x) : (B \to C) \to \big((A \to B) \to (A \to C)\big)}}$$

$\mathrm{comp} : (B \Rightarrow C) \times (A \Rightarrow B) \to (A \Rightarrow C)$

$(g, f) \mapsto g \circ f$

Things we can't do

the λ-calculus

$\lambda x \,.\, f(x) = \big(x \mapsto f(x)\big)$

$$\frac{x \text{ is a variable}}{x \text{ is a program}} \qquad \frac{P \text{ is a program}}{\lambda x \,.\, P \text{ is a program}} \text{ abstraction}$$

extensionality: $f = \big(x \mapsto f(x)\big)$
$P \rightsquigarrow_\eta \lambda x \,.\, P(x)$

$$\frac{P \text{ is a program} \qquad Q \text{ is a program}}{P(Q) \text{ is a program}} \text{ application}$$

evaluating = substituting for bound variable
$P(Q) \rightsquigarrow_\beta P[x \mapsto Q]$
= running the program

A functional programming language lets you
- form functions
- evaluate functions at arguments

Note there's no restrictions on either rule!

$$\cfrac{\cfrac{\cfrac{f \text{ is a variable}}{f \text{ is a program}}}{f(f) \text{ is a program}}}{\lambda f \,.\, f(f) \text{ is a program}}$$

Looping, recursion, …

$$\big(\lambda f \,.\, f(f)\big)\ \big(\lambda f \,.\, f(f)\big) \rightsquigarrow \big(\lambda f \,.\, f(f)\big)\ \big[f \mapsto (\lambda f \,.\, f f)\big]$$
$$= \big(\lambda f \,.\, f(f)\big)\ \big(\lambda f \,.\, f(f)\big)$$

Encode Peano arithmetic

$$1 := \big(\lambda f \,.\, \lambda f \,.\, f(x)\big)$$
$$2 := \big(\lambda f \,.\, \lambda f \,.\, f(f\,x)\big)$$
$$\text{plus} := \big(\lambda m \,.\, \lambda n \,.\, \lambda f \,.\, \lambda x \,.\, m\,f\,(n\,f\,x)\big)$$

Adding primitives

the simply-typed λ-calculus

$\lambda x \,.\, f(x) = \big(x \mapsto f(x)\big)$

A functional programming language lets you
- form functions
- evaluate functions at arguments

$$\frac{}{\underline{n} : \mathrm{nat}} \; (n \in \mathbb{N})$$

$$\frac{}{\mathrm{true} : \mathrm{bool}} \qquad \frac{}{\mathrm{false} : \mathrm{bool}}$$

$$\frac{}{\mathrm{flip}() : \mathrm{bool}}$$

Adding primitives

the simply-typed λ-calculus

$\lambda x\,.\,f(x) = \big(x \mapsto f(x)\big)$

A functional programming language lets you
- form functions
- evaluate functions at arguments

What about plus, if etc?

$$\mathrm{plus} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

$$\mathrm{if} : 2 \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

$$\mathrm{if}(i, n, m) = \begin{cases} n & \text{if } i = 0 \\ m & \text{if } i = 1 \end{cases}$$

$$\frac{}{\underline{n} : \mathrm{nat}}\ (n \in \mathbb{N})$$

$$\frac{}{\mathrm{true} : \mathrm{bool}} \qquad \frac{}{\mathrm{false} : \mathrm{bool}}$$

$$\frac{}{\mathrm{flip}() : \mathrm{bool}}$$

Adding primitives

the simply-typed λ-calculus

$\lambda x \,.\, f(x) = \big(x \mapsto f(x)\big)$

A functional programming language lets you
- form functions
- evaluate functions at arguments

What about plus, if etc?

$$\mathrm{plus} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

$$\mathrm{if} : 2 \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

$$\mathrm{if}(i, n, m) = \begin{cases} n & \text{if } i = 0 \\ m & \text{if } i = 1 \end{cases}$$

$$\frac{}{\underline{n} : \mathrm{nat}} \; (n \in \mathbb{N})$$

$$\frac{}{\mathrm{true} : \mathrm{bool}} \qquad \frac{}{\mathrm{false} : \mathrm{bool}}$$

$$\frac{}{\mathrm{flip}() : \mathrm{bool}}$$

Option 1: $\mathrm{if}(b, n, m) : \mathrm{nat}$ (where $b : \mathrm{bool}, n : \mathrm{nat}, m : \mathrm{nat}$)

Option 2: add a type to model $\mathbb{N} \times \mathbb{N}$

in general: introduce new types for new kinds of structure

Adding product types

the simply-typed λ-calculus

$\lambda x \,.\, f(x) = \big(x \mapsto f(x)\big)$

A functional programming language lets you

- form functions
- evaluate functions at arguments

How does $X \times Y$ behave in Set?

$$\frac{x \in X \qquad\qquad y \in Y}{(x, y) \in X \times Y}\ \text{pair} \qquad\qquad \frac{p \in A_1 \times A_2}{\pi_i(p) \in A_i}\ \text{proj}\ (i = 1,2)$$

project out a pair

$$\pi_i(x_1, x_2) = x_i$$

extensionality: a pair is determined by its projections

$$p = \big(\pi_1(p), \pi_2(p)\big)$$

Adding product types

the simply-typed λ-calculus

$\lambda x \, . \, f(x) = \big(x \mapsto f(x)\big)$

A functional programming language lets you

- form functions
- evaluate functions at arguments

How does $X \times Y$ behave in simply-typed λ-calculus?

$$\frac{P_1 : A_1 \qquad P_2 : A_2}{\langle P_1, P_2 \rangle : A_1 \times A_2} \text{ pair} \qquad\qquad \frac{P : A_1 \times A_2}{\pi_i(P) : A_i} \text{ proj } (i = 1,2)$$

project out a pair

$$\pi_i \langle P_1, P_2 \rangle \rightsquigarrow_\beta P_i$$

extensionality: a pair is determined by its projections

$$P \rightsquigarrow_\eta \langle \pi_1(P), \pi_2(P) \rangle$$

The simply-typed λ-calculus with products and primitives

= the simplest (typed) functional programming language

can also add sums / disjoint unions, lists, recursion, ….

$$\frac{x : A \qquad P : B}{\lambda x \, . \, P : A \to B} \text{ abstraction}$$

$$\frac{P : A \to B \qquad Q : A}{P(Q) : B} \text{ application}$$

evaluating = substituting for bound variable

$$(\lambda x \, . \, P)(Q) \rightsquigarrow_\beta P[x \mapsto Q]$$

= running the program

extensionality: $f = \big(x \mapsto f(x)\big)$

$$P \rightsquigarrow_\eta \lambda x \, . \, P(x)$$

+ any others you might want!

$$\frac{}{\mathrm{true} : \mathrm{bool}} \qquad \frac{}{\mathrm{false} : \mathrm{bool}} \qquad \frac{}{\mathrm{if}(b, n, m) : \mathrm{nat}}$$

$$\frac{}{\mathrm{flip}() : \mathrm{bool}} \qquad \frac{(n \in \mathbb{N})}{\underline{n} : \mathrm{nat}} \qquad \frac{}{\mathrm{plus} : \mathrm{nat} \times \mathrm{nat} \to \mathrm{nat}}$$

$\mathrm{plus}\langle \underline{3}, \underline{2} \rangle : \mathrm{nat}$

$\mathrm{if}(\mathrm{true}, \underline{3}, \underline{2}) : \mathrm{nat}$

$$\frac{P_1 : A_1 \qquad P_2 : A_2}{\langle P_1, P_2 \rangle : A_1 \times A_2} \text{ pair}$$

project out a pair

$$\pi_i(P_1, P_2) \rightsquigarrow_\beta P_i$$

$$\frac{P : A_1 \times A_2}{\pi_i(P) : A_i} \text{ proj } (i = 1{,}2)$$

extensionality: a pair is determined by its projections

$$P \rightsquigarrow_\eta \big(\pi_1(P), \pi_2(P)\big)$$

+ a ‘unit’ type

β-reduction = running the program

$$
\begin{aligned}
\big(\lambda p : \mathrm{nat} \times \mathrm{nat} \to \mathrm{bool}\,.\, \lambda t : \mathrm{nat} \times \mathrm{nat}\,.\, \mathrm{if}(p(t), \underline{2}, \underline{3})\big)&(\mathrm{greater\_than})(\langle \underline{5}, \underline{6} \rangle) \\
&\rightsquigarrow_\beta \big(\lambda t : \mathrm{nat} \times \mathrm{nat}\,.\, \mathrm{if}(\mathrm{greater\_than}(t), \underline{2}, \underline{3})\big)(\langle \underline{5}, \underline{6} \rangle) \\
&\rightsquigarrow_\beta \mathrm{if}\big(\mathrm{greater\_than}\langle \underline{5}, \underline{6} \rangle, \underline{2}, \underline{3}\big) \\
&\rightsquigarrow_\beta \underline{3}
\end{aligned}
$$

The magic of higher-order functions

higher-order functions = functions of type $(A \to B) \to C$

higher-order functions let you re-use code in a very efficient way

$$P : \big((\mathrm{nat} \to \mathrm{bool}) \times (\mathrm{nat} \to \mathrm{nat})\big) \to \mathrm{nat}$$

acts on an arbitrary predicate and arbitrary endo-function on nat

$$\begin{aligned} \mathrm{eval} : (A \Rightarrow B) \times A &\to B \\ (f, x) &\mapsto f(x) \end{aligned}$$

$$\lambda p \,.\, \big(\pi_1(p)\big)\big(\pi_2(p)\big) : (A \to B) \times A \to B$$

$\pi_1(p) : (A \to C)$
$\pi_2(p) : A$

$$\begin{aligned} \mathrm{comp} : (B \Rightarrow C) \times (A \Rightarrow B) &\to (A \Rightarrow C) \\ (g, f) &\mapsto g \circ f \end{aligned}$$

$$\lambda f \,.\, \lambda x \,.\, \big(\pi_1(f)\big)\big(\pi_2(f)(x)\big) : \big((B \to C) \times (A \to B)\big) \to (A \to C)$$

$\pi_1(f) : (B \to C)$
$\pi_2(f) : (A \to B)$
$\pi_2(f)(x) : B$
$\big(\pi_1(f)\big)\big(\pi_2(f)(x)\big) : C$
$\lambda x \,.\, \big(\pi_1(f)\big)\big(\pi_2(f)(x)\big) : A \to C$

Note the observable behaviour is about when values get returned

this is what we care about!

What does programming language theory study?

We want programs that are:

efficient, fast, and correct

We ask:

(1) How should we think about programs?

terms in some version of simply-typed λ-calculus

(2) When are programs interchangeable?

Two notions of equality:

(1) "equality as functions"

(2) "equality as programs"

= same behaviour no matter what program you put them into

What does programming language theory study?

We want programs that are:

efficient, fast, and correct

We ask:

(1) How should we think about programs?

terms in some version of simply-typed λ-calculus

(2) When are programs interchangeable?

$$(\lambda x \,.\, P)(Q) =_{\beta\eta} P[x \mapsto Q]$$
$$P =_{\beta\eta} \lambda x \,.\, P(x)$$
$$\pi_i(\langle P_1, P_2 \rangle) =_{\beta\eta} P_i \qquad (i = 1,2)$$
$$P =_{\beta\eta} \langle \pi_1(P), \pi_2(P) \rangle$$

Two notions of equality:

βη-equality $=_{\beta\eta}$: the congruence generated by $\rightsquigarrow_\beta \cup \rightsquigarrow_\eta$

observational equivalence: $P \simeq_{\text{obs}} Q$ iff whatever program $C[\_]$ of type bool or nat we put them in, $C[P]$ and $C[Q]$ have the same behaviour

$P =_{\beta\eta} Q \implies P \simeq_{\text{ctx}} Q$

converse is false!

for every program $C[-]$ with a 'hole' such that $C[P], C[Q] : \mathrm{nat}$ or $C[P], C[Q] : \mathrm{bool}$, we have $C[P]$ terminates with output V and effect $E \iff C[Q]$ terminates with output V and effect E

What does programming language theory study?

We want programs that are:

efficient, fast, and correct

We ask:

(1) How should we think about programs?

terms in some version of simply-typed λ-calculus

(2) When are programs interchangeable?

Two notions of equality:

βη-equality $=_{\beta\eta}$: the congruence generated by $\rightsquigarrow_\beta \cup \rightsquigarrow_\eta$

$$(\lambda x \,.\, P)(Q) =_{\beta\eta} P[x \mapsto Q]$$
$$P =_{\beta\eta} \lambda x \,.\, P(x)$$
$$\pi_i(\langle P_1, P_2 \rangle) =_{\beta\eta} P_i \qquad (i = 1,2)$$
$$P =_{\beta\eta} \langle \pi_1(P), \pi_2(P) \rangle$$

observational equivalence: $P \simeq_{\text{obs}} Q$ iff whatever program $C[\_]$ of type bool or nat we put them in, $C[P]$ and $C[Q]$ have the same behaviour

$P =_{\beta\eta} Q \implies P \simeq_{\text{ctx}} Q$

converse is false!

Note the observable behaviour is about when values get returned

this is what we care about!

for every program $C[-]$ with a 'hole' such that $C[P], C[Q] : \mathrm{nat}$ or $C[P], C[Q] : \mathrm{bool}$, we have $C[P]$ terminates with output V and effect E $\iff$ $C[Q]$ terminates with output V and effect E

What does programming language theory study?

We ask:

(1) How should we think about programs?

terms in some version of simply-typed λ-calculus

(2) When are programs interchangeable?

Two notions of equality:

βη-equality $=_{\beta\eta}$: the congruence generated by $\rightsquigarrow_\beta \cup \rightsquigarrow_\eta$

observational equivalence: $P \simeq_{\text{obs}} Q$ iff whatever program $C[\_]$ of type bool or nat we put them in, $C[P]$ and $C[Q]$ have the same behaviour

Note the observable behaviour is about when values get returned

this is what we care about!

Two schools:

(1) Syntactic techniques

(2) Semantic techniques

What does programming language theory study?

We ask:

(1) How should we think about programs?

terms in some version of simply-typed λ-calculus

(2) When are programs interchangeable?

Two notions of equality:

βη-equality $=_{\beta\eta}$: the congruence generated by $\rightsquigarrow_\beta \cup \rightsquigarrow_\eta$

observational equivalence: $P \simeq_{\text{obs}} Q$ iff whatever program $C[\_]$ of type bool or nat we put them in, $C[P]$ and $C[Q]$ have the same behaviour

Note the observable behaviour is about when values get returned

this is what we care about!

Two schools:

(1) Syntactic techniques

use the syntax and the $\rightsquigarrow$ relations directly; generally inductive arguments

(2) Semantic techniques

easy to refute observational equivalences; hard to prove them!

What does programming language theory study?

We ask:

(1) How should we think about programs?

Terms in some version of simply-typed λ-calculus

(2) When are programs interchangeable?

Two notions of equality:

βη-equality $=_{\beta\eta}$: the congruence generated by $\rightsquigarrow_\beta \cup \rightsquigarrow_\eta$

observational equivalence: $P \simeq_{\text{obs}} Q$ iff whatever program $C[\_]$ of type bool or nat we put them in, $C[P]$ and $C[Q]$ have the same behaviour

Note the observable behaviour is about when values get returned

this is what we care about!

Two schools:

(1) Syntactic techniques

use the syntax and the $\rightsquigarrow$ relations directly; generally inductive arguments

easy to refute observational equivalences; hard to prove them!

(2) Semantic techniques

build semantic models and study those instead

easier to prove observational equivalences; hard to refute them!

Coming up next

1. Introduce an idealised functional programming language
2. Explain its semantic interpretation in CCCs

---

3. Introduce differentiable programming
4. Explain the interpretation in Diff

Cartesian closed categories (CCCs)

def: a cartesian closed category $(\mathbb{C}, \times, 1, \Rightarrow)$ is a category $\mathbb{C}$

with finite products $(\times, 1)$

and a right adjoint $A \Rightarrow (-)$ for every $(-) \times A$

$$\mathbb{C}(X, A_1 \times A_2) \cong \mathbb{C}(X, A_1) \times \mathbb{C}(X, A_2)$$

$$f \mapsto (\pi_1 \circ f, \pi_2 \circ f)$$

$$\langle f_1, f_2 \rangle \leftarrow\!\shortmid (f_1, f_2)$$

$\langle f_1, f_2 \rangle(x) = (f_1 x, f_2 x)$

$$\mathbb{C}(X \times A, B) \cong \mathbb{C}(X, A \Rightarrow B)$$

$$f \mapsto \Lambda(f)$$ $\Lambda(f)(x) = f(x, \_)$

$$\mathrm{eval} \circ (f \times A) \leftarrow\!\shortmid f$$

$\tilde{f}(x, a) = f(x)(a)$

Cartesian closed categories (CCCs)

def: a cartesian closed category $(\mathbb{C}, \times, 1, \Rightarrow)$ is a category $\mathbb{C}$

with finite products $(\times, 1)$

and a right adjoint $A \Rightarrow (-)$ for every $(-) \times A$

$$\mathbb{C}(X, A_1 \times A_2) \cong \mathbb{C}(X, A_1) \times \mathbb{C}(X, A_2)$$

$$f \mapsto (\pi_1 \circ f, \pi_2 \circ f)$$

$$\langle f_1, f_2 \rangle \leftarrow\!\shortmid (f_1, f_2)$$

$\langle f_1, f_2 \rangle = \lambda x\,.\,(f_1 x, f_2 x)$

$$\mathbb{C}(X \times A, B) \cong \mathbb{C}(X, A \Rightarrow B)$$

$$f \mapsto \Lambda(f) \quad \Lambda(f) = \lambda x\,.\,f(x, \_)$$

$$\mathrm{eval} \circ (f \times A) \leftarrow\!\shortmid f$$

$\mathrm{eval} = \lambda(f, x)\,.\,f(x)$

$\mathrm{eval} \circ (f \times A) = \lambda(x, a)\,.\,f(x)(a)$

Cartesian closed categories (CCCs)

def: a cartesian closed category $(\mathbb{C}, \times, 1, \Rightarrow)$ is a category $\mathbb{C}$

with finite products $(\times, 1)$

and a right adjoint $A \Rightarrow (-)$ for every $(-) \times A$

$$\mathbb{C}(X, A_1 \times A_2) \cong \mathbb{C}(X, A_1) \times \mathbb{C}(X, A_2)$$

$$f \mapsto (\pi_1 \circ f, \pi_2 \circ f)$$ projections

pairing $$\langle f_1, f_2 \rangle \leftarrow\!\shortmid (f_1, f_2)$$

$\langle f_1, f_2 \rangle = \lambda x \,.\, (f_1 x, f_2 x)$

$$\mathbb{C}(X \times A, B) \cong \mathbb{C}(X, A \Rightarrow B)$$

$$f \mapsto \Lambda(f)$$ $\Lambda(f) = \lambda x \,.\, f(x, \_)$

abstraction

$$\mathrm{eval} \circ (f \times A) \leftarrow\!\shortmid f$$

$\mathrm{eval} = \lambda(f, x) \,.\, f(x)$ application

$\mathrm{eval} \circ (f \times A) = \lambda(x, a) \,.\, f(x)(a)$

Semantic interpretation

simply-typed λ-calculus	$\xrightarrow[\text{semantic interpretation}]{[\![\_]\!]}$	cartesian closed category $\mathbb{C}$
type A	$\longrightarrow$	object $[\![A]\!]$
product type exponential type		product object exponential object
program $P : A$	$\longrightarrow$	morphism with codomain $[\![A]\!]$
pairing projections		pairing projections
abstraction application		currying $\Lambda(\_)$ evaluating at arguments

Meanings for types in a CCC

$$\text{Types} \ni A, B ::= \text{nat} \mid \text{bool} \mid A \times B \mid A \to B$$

Meanings for types in a CCC

$$\text{Types} \ni A, B ::= \text{nat} \mid \text{bool} \mid A \times B \mid A \to B$$

$$[\![\text{nat}]\!] := \mathbb{N}$$
$$[\![\text{bool}]\!] := 2$$

chosen objects

eg $2 := 1 + 1$, $\mathbb{N}$:= a natural numbers object

Meanings for types in a CCC

$$\text{Types} \ni A, B ::= \text{nat} \mid \text{bool} \mid A \times B \mid A \to B$$

$$\begin{aligned}
[\![\text{nat}]\!] &:= \mathbb{N} \\
[\![\text{bool}]\!] &:= 2 \\
[\![A \times B]\!] &:= [\![A]\!] \times [\![B]\!] \\
[\![A \to B]\!] &:= \left([\![A]\!] \Rightarrow [\![B]\!]\right)
\end{aligned}$$

chosen objects

eg $2 := 1 + 1$, $\mathbb{N}$:= a natural numbers object

$$\begin{aligned}
[\![\text{nat} \to \text{bool}]\!] &:= (\mathbb{N} \Rightarrow 2) \\
[\![\text{bool} \to \text{bool}]\!] &:= (2 \Rightarrow 2) \\
&\vdots
\end{aligned}$$

Meanings for terms in a CCC

handling free variables

no free variables

$\mathrm{plus} : \mathrm{nat} \times \mathrm{nat} \to \mathrm{nat}$

assigns something of type nat whenever we give $P : \mathrm{nat} \times \mathrm{nat}$

so $[\![\mathrm{plus}]\!]$ is a map $[\![\mathrm{nat}]\!] \times [\![\mathrm{nat}]\!] \to [\![\mathrm{nat}]\!]$;
equivalently, a map $1 \to \big(([\![\mathrm{nat}]\!] \times [\![\mathrm{nat}]\!]) \Rightarrow [\![\mathrm{nat}]\!]\big)$

Meanings for terms in a CCC

handling free variables

no free variables

$\mathrm{plus} : \mathrm{nat} \times \mathrm{nat} \to \mathrm{nat}$

assigns something of type nat whenever we give $P : \mathrm{nat} \times \mathrm{nat}$

so $[\![\mathrm{plus}]\!]$ is a map $[\![\mathrm{nat}]\!] \times [\![\mathrm{nat}]\!] \to [\![\mathrm{nat}]\!]$;
equivalently, a map $1 \to \big(([\![\mathrm{nat}]\!] \times [\![\mathrm{nat}]\!]) \Rightarrow [\![\mathrm{nat}]\!]\big)$

b, n and m free

$\mathrm{if}(b, n, m) : \mathrm{nat}$

assigns something of type nat whenever we give $b : \mathrm{bool}, n : \mathrm{nat}$ and $m : \mathrm{nat}$

so $[\![\mathrm{if}(b, n, m)]\!]$ is a map $[\![\mathrm{bool}]\!] \times [\![\mathrm{nat}]\!] \times [\![\mathrm{nat}]\!] \to [\![\mathrm{nat}]\!]$

Meanings for terms in a CCC

handling free variables

no free variables

$\mathrm{plus} : \mathrm{nat} \times \mathrm{nat} \to \mathrm{nat}$

assigns something of type nat whenever we give $P : \mathrm{nat} \times \mathrm{nat}$

so $[\![\mathrm{plus}]\!]$ is a map $[\![\mathrm{nat}]\!] \times [\![\mathrm{nat}]\!] \to [\![\mathrm{nat}]\!]$;
equivalently, a map $1 \to \big(([\![\mathrm{nat}]\!] \times [\![\mathrm{nat}]\!]) \Rightarrow [\![\mathrm{nat}]\!]\big)$

b, n and m free

$\mathrm{if}(b, n, m) : \mathrm{nat}$

assigns something of type nat whenever we give $b : \mathrm{bool}, n : \mathrm{nat}$ and $m : \mathrm{nat}$

so $[\![\mathrm{if}(b, n, m)]\!]$ is a map $[\![\mathrm{bool}]\!] \times [\![\mathrm{nat}]\!] \times [\![\mathrm{nat}]\!] \to [\![\mathrm{nat}]\!]$

$P : B$ with free variables $(x_i : A_i)_{i=1,\ldots,n}$ has interpretation $[\![P]\!] : \prod_{i=1}^{n} [\![A_i]\!] \to [\![B]\!]$

assigns $a_i \in [\![A_i]\!]$ to each $x_i : A_i$

eg $[\![\mathrm{if}]\!](0,2,3) = 2$

Meanings for terms in a CCC

handling free variables

no free variables

$\mathrm{plus} : \mathrm{nat} \times \mathrm{nat} \to \mathrm{nat}$

assigns something of type nat whenever we give $P : \mathrm{nat} \times \mathrm{nat}$

so $[\![\mathrm{plus}]\!]$ is a map $[\![\mathrm{nat}]\!] \times [\![\mathrm{nat}]\!] \to [\![\mathrm{nat}]\!]$;
equivalently, a map $1 \to \big(([\![\mathrm{nat}]\!] \times [\![\mathrm{nat}]\!]) \Rightarrow [\![\mathrm{nat}]\!]\big)$

b, n and m free

$\mathrm{if}(b, n, m) : \mathrm{nat}$

assigns something of type nat whenever we give $b : \mathrm{bool}, n : \mathrm{nat}$ and $m : \mathrm{nat}$

so $[\![\mathrm{if}(b, n, m)]\!]$ is a map $[\![\mathrm{bool}]\!] \times [\![\mathrm{nat}]\!] \times [\![\mathrm{nat}]\!] \to [\![\mathrm{nat}]\!]$

$P : B$ with free variables $(x_i : A_i)_{i=1,\ldots,n}$ has
interpretation $[\![P]\!] : \prod_{i=1}^{n} [\![A_i]\!] \to [\![B]\!]$

assigns $a_i \in [\![A_i]\!]$ to each $x_i : A_i$
eg $[\![\mathrm{if}]\!](0{,}2{,}3) = 2$

for $P : B$ with no free variables,
$[\![P]\!] : 1 \to [\![B]\!]$

so eg $P : A \to B$ is identified with an element of $([\![A]\!] \Rightarrow [\![B]\!])$

Meanings for closed terms in $\mathcal{S}\mathrm{et}$

$P : B$ with free variables $(x_i : A_i)_{i=1,\ldots,n}$
has interpretation
$[\![P]\!] : \prod_{i=1}^{n} [\![A_i]\!] \to [\![B]\!]$

assigns $a_i \in [\![A_i]\!]$ to each $x_i : A_i$

Fix an interpretation $[\![c]\!]$ for each primitive c

For $P : B$ with no free variables, $[\![P]\!] \in [\![B]\!]$:

$$[\![\pi_i(P)]\!] = \Big(i\text{th projection out } [\![P]\!] \Big) \in [\![B_i]\!]$$

$$[\![\langle P_1, P_2 \rangle]\!] = \Big([\![P_1]\!] \quad , [\![P_2]\!] \Big) \in [\![B_1]\!] \times [\![B_2]\!]$$

$$[\![P(Q)]\!] = \Big([\![P]\!] \Big)\Big([\![Q]\!] \Big) \in [\![C]\!]$$

$$[\![\lambda x \, . \, P]\!] = \lambda b \, . \, [\![P]\!](\quad b) \in [\![B]\!] \Rightarrow [\![C]\!]$$

Meanings for terms in $\mathcal{S}\mathrm{et}$

$P : B$ with free variables $(x_i : A_i)_{i=1,\ldots,n}$
has interpretation
$[\![P]\!] : \prod_{i=1}^{n} [\![A_i]\!] \to [\![B]\!]$

assigns $a_i \in [\![A_i]\!]$ to each $x_i : A_i$

Fix an interpretation $[\![c]\!]$ for each primitive c

For $\overrightarrow{a} \in \prod_{i=1}^{n} [\![A_i]\!]$

assigning $a_i \in [\![A_i]\!]$ to each free $x_i : A_i$ in P:

Meanings for terms in Set

$P : B$ with free variables $(x_i : A_i)_{i=1,\ldots,n}$
has interpretation
$[\![P]\!] : \prod_{i=1}^{n} [\![A_i]\!] \to [\![B]\!]$

assigns $a_i \in [\![A_i]\!]$ to each $x_i : A_i$

Fix an interpretation $[\![c]\!]$ for each primitive c

For $\overrightarrow{a} \in \prod_{i=1}^{n} [\![A_i]\!]$
assigning $a_i \in [\![A_i]\!]$ to each free $x_i : A_i$ in P:

$$[\![\pi_i(P)]\!](\overrightarrow{a}) = \big(i\text{th projection out } [\![P]\!](\overrightarrow{a})\big) \quad \in [\![B_i]\!]$$

$$[\![\langle P_1, P_2\rangle]\!](\overrightarrow{a}) = \big([\![P_1]\!](\overrightarrow{a}), [\![P_2]\!](\overrightarrow{a})\big) \quad \in [\![B_1]\!] \times [\![B_2]\!]$$

Meanings for terms in $\mathbb{S}\mathrm{et}$

$P : B$ with free variables $(x_i : A_i)_{i=1,\ldots,n}$
has interpretation
$[\![P]\!] : \prod_{i=1}^{n} [\![A_i]\!] \to [\![B]\!]$

assigns $a_i \in [\![A_i]\!]$ to each $x_i : A_i$

Fix an interpretation $[\![c]\!]$ for each primitive c

For $\overrightarrow{a} \in \prod_{i=1}^{n} [\![A_i]\!]$

assigning $a_i \in [\![A_i]\!]$ to each free $x_i : A_i$ in P:

$$[\![\pi_i(P)]\!](\overrightarrow{a}) = \big(i\text{th projection out } [\![P]\!](\overrightarrow{a})\big) \quad \in [\![B_i]\!]$$

$$[\![\langle P_1, P_2\rangle]\!](\overrightarrow{a}) = \big([\![P_1]\!](\overrightarrow{a}), [\![P_2]\!](\overrightarrow{a})\big) \quad \in [\![B_1]\!] \times [\![B_2]\!]$$

$$[\![P(Q)]\!](\overrightarrow{a}) = \big([\![P]\!](\overrightarrow{a})\big)\,\big([\![Q]\!](\overrightarrow{a})\big) \quad \in [\![C]\!]$$

$$[\![\lambda x\,.\,P]\!](\overrightarrow{a}) = \lambda b\,.\,[\![P]\!](\overrightarrow{a}, b) \quad \in [\![B]\!] \Rightarrow [\![C]\!]$$

Meanings for terms in a CCC

$P : B$ with free variables $(x_i : A_i)_{i=1,\ldots,n}$
has interpretation
$[\![P]\!] : \prod_{i=1}^{n} [\![A_i]\!] \to [\![B]\!]$

assigns $a_i \in [\![A_i]\!]$ to each $x_i : A_i$

Fix an interpretation $[\![c]\!]$ for each primitive c

Then:

$$[\![\pi_i(P) : B_i]\!] = \pi_i \circ [\![P : B_1 \times B_2]\!]$$

$$[\![\langle P_1, P_2 \rangle : B_1 \times B_2]\!] = \Big\langle [\![P_1 : B_1]\!], [\![P_2 : B_2]\!] \Big\rangle$$

$\langle f_1, f_2 \rangle(x) = (f_1 x, f_2 x)$

$$[\![P(Q) : C]\!] = \text{eval} \circ \langle [\![P : B \to C]\!], [\![Q : B]\!] \rangle$$

$$[\![\lambda x \,.\, P : B \to C]\!] = \Lambda([\![P : C]\!])$$

$\Lambda(f) = \lambda x \,.\, f(x, \_)$

$\text{eval} \circ (f \times A) = \lambda(x, a) \,.\, f(x)(a)$

Soundness of the interpretation

$P : B$ with free variables $(x_i : A_i)_{i=1,\ldots,n}$
has interpretation
$[\![P]\!] : \prod_{i=1}^{n} [\![A_i]\!] \to [\![B]\!]$

assigns $a_i \in [\![A_i]\!]$ to each $x_i : A_i$

for any CCC $\mathbb{C}$ and any choice of base types and constants,

$$P =_{\beta\eta} Q \implies [\![P]\!] = [\![Q]\!]$$

in an adequate model, $[\![P]\!] = [\![Q]\!] \implies P \simeq_{\text{obs}} Q$

in fact, simply-typed λ-calculus modulo $=_{\beta\eta}$ is
a sound and complete logic for CCCs

What does programming language theory study?

We want programs that are:

efficient, fast, and correct

We ask:

(1) How should we think about programs?

(2) When are programs interchangeable?

Two notions of equality:

βη-equality $=_{\beta\eta}$: the congruence generated by $\rightsquigarrow_\beta \cup \rightsquigarrow_\eta$

observational equivalence: $P \simeq_{\text{obs}} Q$ iff whatever program $C[\_]$ of type bool or nat we put them in, $C[P]$ and $C[Q]$ have the same behaviour

What does denotational semantics study?

We want programs that are:

efficient, fast, and correct

We ask:

(1) How should we think about programs?

(2) When are programs interchangeable?

terms in some version of simply-typed λ-calculus interpreted in a CCC

Two notions of equality:

βη-equality $=_{\beta\eta}$: the congruence generated by $\rightsquigarrow_\beta \cup \rightsquigarrow_\eta$

observational equivalence: $P \simeq_{\text{obs}} Q$ iff whatever program $C[\_]$ of type bool or nat we put them in, $C[P]$ and $C[Q]$ have the same behaviour

use adequate models to reason about observational equivalence of programs

adequacy: $[\![P]\!] = [\![Q]\!] \implies P \simeq_{\text{obs}} Q$

Some example interpretations

languages with no effects

plain CCCs

languages with printing, global memory, exceptions

CCCs with a (strong) monad

the monad T describes the effect, eg $(-)+1$ or $S^* \times (-)$

languages with local memory

presheaf categories

think: programs parametrised by possible states of the memory

languages with recursion

order-enriched categories

$\leq$ models 'how defined' a function is

each recursive call goes up the order; the whole loop is then a fixpoint

looping forever modelled by a bottom element

How should we think about programs?

```
fun add(x, y):
   return (x + y)
```

a function $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$

```
fun divide(x, y):
   return (x / y)
```

a function $\mathbb{Z} \times \mathbb{Z}_{\neq 0} \to \mathbb{Q}$

a function $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Q} + \{\text{fail}\}$

```
fun print_and_return(x):
  print “hello”;
  return x;
```

a function $\mathbb{N} \to \{a, b, \ldots, z\}^* \times \mathbb{N}$

$x \mapsto (\text{hello}, x)$

```
let b = flip(p);
return b;
```

a probability distribution on $\{\text{true}, \text{false}\}$

```
normalise(
  let x = sample(bernoulli(0.8));
  let r = (if x then 10 else 3);
  observe 0.45 from exponential(r)
  return(x)
)
```

some measurable function (??)

Coming up next

1. Introduce an idealised functional programming language
2. Explain its semantic interpretation in CCCs

---

3. Introduce differentiable programming
4. Explain the interpretation in Diff

high-dimensional input

low-dimensional output

cat

dog

high-dimensional input $\xrightarrow{\text{program } P \text{ with many parameters}}$ low-dimensional output

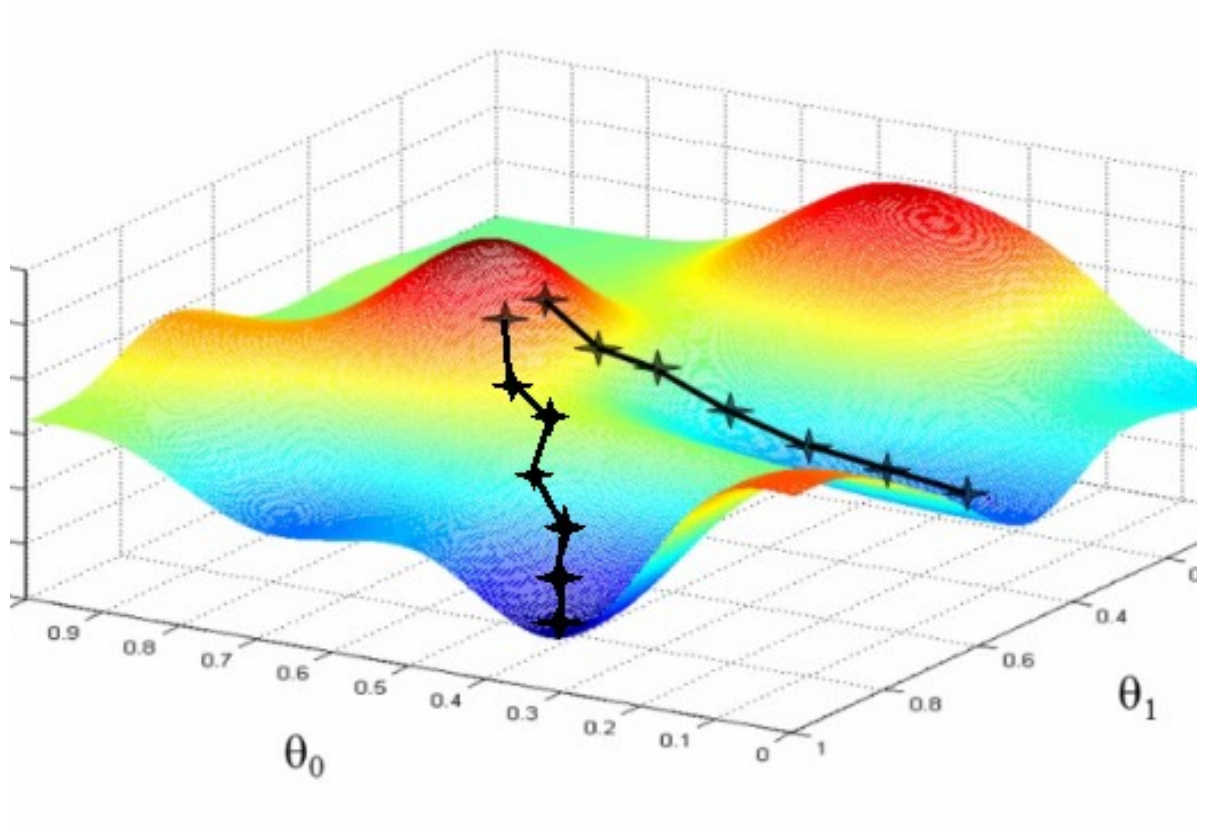
eg a neural network with many layers, and different weights for the activation functions

high-dimensional input

program P with many parameters

eg a neural network with many layers, and different weights for the activation functions

low-dimensional output



https://www.youtube.com/watch?v=5u4G23_OohI

ie differentiate the function described by P

aim: optimise the parameters for P

so that, eg, it classifies cats as cats as often as possible

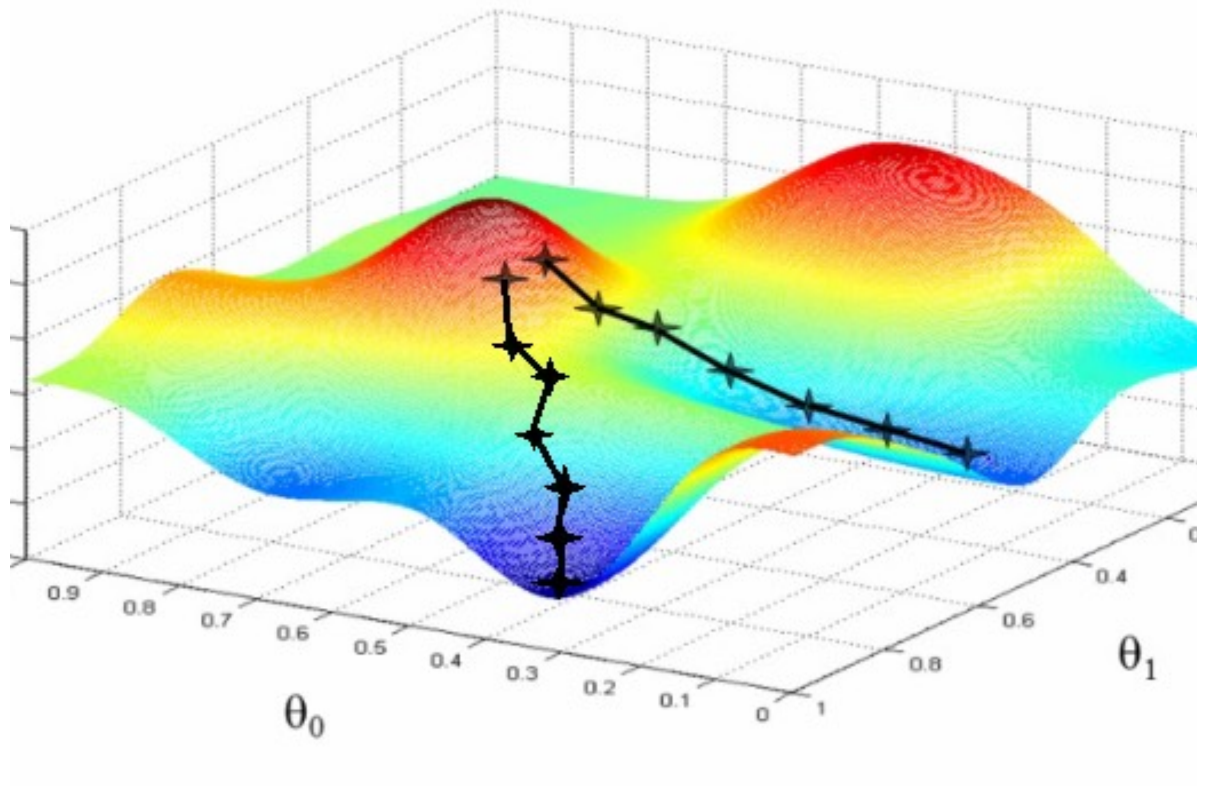
can be done numerically, but it's hard in general!

high-dimensional input

program P with many parameters

eg a neural network with many layers, and different weights for the activation functions

low-dimensional output



https://www.youtube.com/watch?v=5u4G23_OohI

ie differentiate the function described by P

aim: optimise the parameters for P

so that, eg, it classifies cats as cats as often as possible

can be done numerically, but it's hard in general!

can we write an algorithm to calculate derivatives exactly?

…and can we prove this is correct?

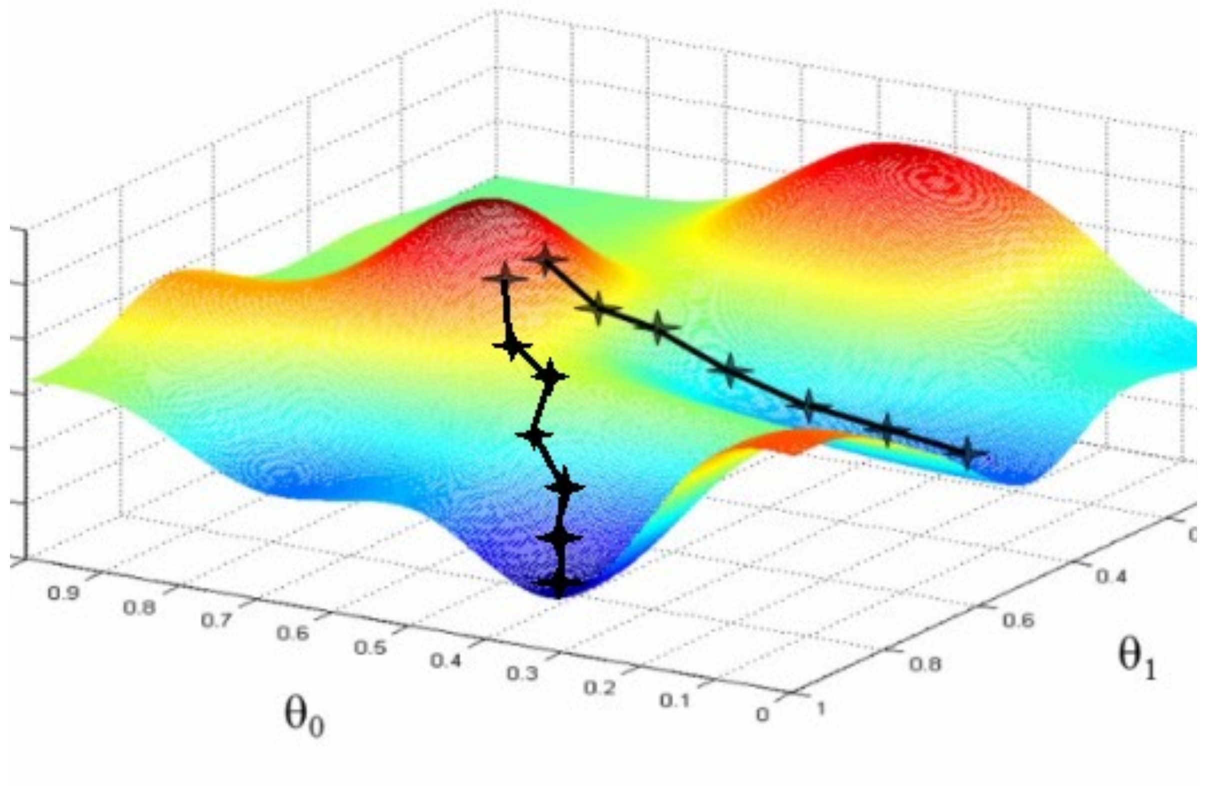
"forward AD", "reverse AD", etc

high-dimensional input

program P with many parameters

eg a neural network with many layers, and different weights for the activation functions

low-dimensional output



https://www.youtube.com/watch?v=5u4G23_OohI

ie differentiate the function described by P

aim: optimise the parameters for P

so that, eg, it classifies cats as cats as often as possible

can be done numerically, but it's hard in general!

can we write an algorithm to calculate derivatives exactly?

…and can we prove this is correct?

"forward AD", "reverse AD", etc

differentiable programming (TensorFlow, PyTorch, etc)

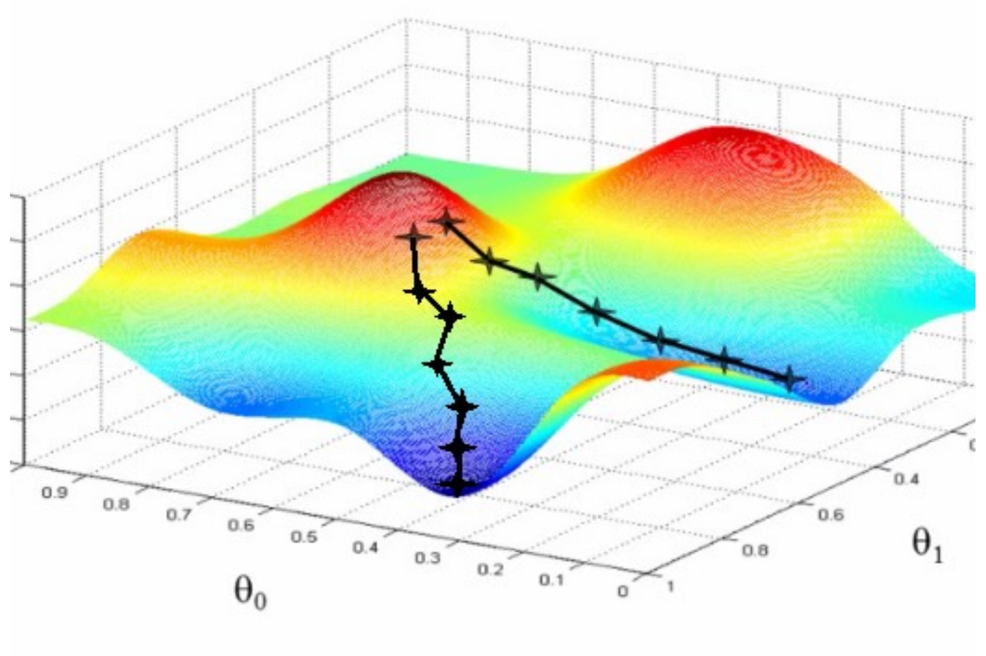
= languages where you can automatically compute the derivative of any program

high-dimensional input

program P with many parameters

eg a neural network with many layers, and different weights for the activation functions

low-dimensional output



https://www.youtube.com/watch?v=5u4G23_OohI

ie differentiate the function described by P

aim: optimise the parameters for P

so that, eg, it classifies cats as cats as often as possible

can be done numerically, but it's hard in general!

= a serious bottleneck

from the denotational semantics POV:

(1) $[\![P]\!]$ is some smooth function $\mathbb{R}^n \to \mathbb{R}$

(2) aim: to algorithmically define a program $D(P)$

and prove that $[\![D(P)]\!] = D([\![P]\!])$

essentially, using the chain rule and "dual numbers"

Proving correctness of automatic differentiation

[Huot, Staton, Vakar]

a natural suggestion:

(1) we only care about the programs returning a value, ie those of type real

(2) take simply-typed λ-calculus + primitives for real numbers etc

(3) a program $P : \mathrm{real}$ is meant to represent a smooth function $\mathbb{R}^n \to \mathbb{R}$

(4) define $D(P)$ by induction on the simply-typed λ-calculus
and check $[\![D(P)]\!] = D([\![P]\!])$

Proving correctness of automatic differentiation

[Huot, Staton, Vakar]

a natural suggestion:

(1) we only care about the programs returning a value, ie those of type real

(2) take simply-typed λ-calculus + primitives for real numbers etc

(3) a program $P : \mathrm{real}$ is meant to represent a smooth function $\mathbb{R}^n \to \mathbb{R}$

(4) define $D(P)$ by induction on the simply-typed λ-calculus

and check $[\![D(P)]\!] = D([\![P]\!])$

Proving correctness of automatic differentiation

[Huot, Staton, Vakar]

a natural suggestion:

(1) we only care about the programs returning a value, ie those of type real

(2) take simply-typed λ-calculus + primitives for real numbers etc

(3) a program $P : \mathrm{real}$ is meant to represent a smooth function $\mathbb{R}^n \to \mathbb{R}$

(4) define $D(P)$ by induction on the simply-typed λ-calculus

and check $[\![D(P)]\!] = D([\![P]\!])$

Proving correctness of automatic differentiation

[Huot, Staton, Vakar]

a natural suggestion:

(1) we only care about the programs returning a value, ie those of type real

(2) take simply-typed λ-calculus + primitives for real numbers etc

(3) a program $P : \mathrm{real}$ is meant to represent a smooth function $\mathbb{R}^n \to \mathbb{R}$

(4) define $D(P)$ by induction on the simply-typed λ-calculus

and check $[\![D(P)]\!] = D([\![P]\!])$

Proving correctness of automatic differentiation

[Huot, Staton, Vakar]

a natural suggestion:

(1) we only care about the programs returning a value, ie those of type real

(2) take simply-typed λ-calculus + primitives for real numbers etc

(3) a program $P : \mathrm{real}$ is meant to represent a smooth function $\mathbb{R}^n \to \mathbb{R}$

(4) define $D(P)$ by induction on the simply-typed λ-calculus

and check $[\![D(P)]\!] = D([\![P]\!])$

Proving correctness of automatic differentiation

[Huot, Staton, Vakar]

a natural suggestion:

(1) we only care about the programs returning a value, ie those of type real

(2) take simply-typed λ-calculus + primitives for real numbers etc

(3) a program $P : \mathrm{real}$ is meant to represent a smooth function $\mathbb{R}^n \to \mathbb{R}$

(4) define $D(P)$ by induction on the simply-typed λ-calculus

and check $[\![D(P)]\!] = D([\![P]\!])$

$$\overrightarrow{\mathcal{D}}(x) \stackrel{\text{def}}{=} x \qquad \overrightarrow{\mathcal{D}}(\underline{c}) \stackrel{\text{def}}{=} \langle \underline{c}, 0 \rangle$$

$$\overrightarrow{\mathcal{D}}(t+s) \stackrel{\text{def}}{=} \mathbf{case}\, \overrightarrow{\mathcal{D}}(t)\, \mathbf{of}\, \langle x, x' \rangle \to \mathbf{case}\, \overrightarrow{\mathcal{D}}(s)\, \mathbf{of}\, \langle y, y' \rangle \to \langle x + y, x' + y' \rangle$$

$$\overrightarrow{\mathcal{D}}(t * s) \stackrel{\text{def}}{=} \mathbf{case}\, \overrightarrow{\mathcal{D}}(t)\, \mathbf{of}\, \langle x, x' \rangle \to \mathbf{case}\, \overrightarrow{\mathcal{D}}(s)\, \mathbf{of}\, \langle y, y' \rangle \to \langle x * y, x * y' + x' * y \rangle$$

$$\overrightarrow{\mathcal{D}}(\varsigma(t)) \stackrel{\text{def}}{=} \mathbf{case}\, \overrightarrow{\mathcal{D}}(t)\, \mathbf{of}\, \langle x, x' \rangle \to \mathbf{let}\, y = \varsigma(x)\, \mathbf{in}\, \langle y, x' * y * (1 - y) \rangle$$

$$\overrightarrow{\mathcal{D}}(\lambda x.t) \stackrel{\text{def}}{=} \lambda x.\overrightarrow{\mathcal{D}}(t) \qquad \overrightarrow{\mathcal{D}}(t\, s) \stackrel{\text{def}}{=} \overrightarrow{\mathcal{D}}(t)\, \overrightarrow{\mathcal{D}}(s) \qquad \overrightarrow{\mathcal{D}}(\langle t_1, \ldots, t_n \rangle) \stackrel{\text{def}}{=} \langle \overrightarrow{\mathcal{D}}(t_1), \ldots, \overrightarrow{\mathcal{D}}(t_n) \rangle$$

from *Correctness of Automatic Differentiation via Diffeologies and Categorical Gluing*

Proving correctness of automatic differentiation

[Huot, Staton, Vakar]

a natural suggestion:

(1) we only care about the programs returning a value, ie those of type real

(2) take simply-typed λ-calculus + primitives for real numbers etc

(3) a program $P : \mathrm{real}$ is meant to represent a smooth function $\mathbb{R}^n \to \mathbb{R}$

(4) define $D(P)$ by induction on the simply-typed λ-calculus

and check $[\![D(P)]\!] = D([\![P]\!])$

ie. we interpret in the category of cartesian spaces (= $\mathbb{R}^n$ for some n) and smooth maps

but this category is not cartesian closed!

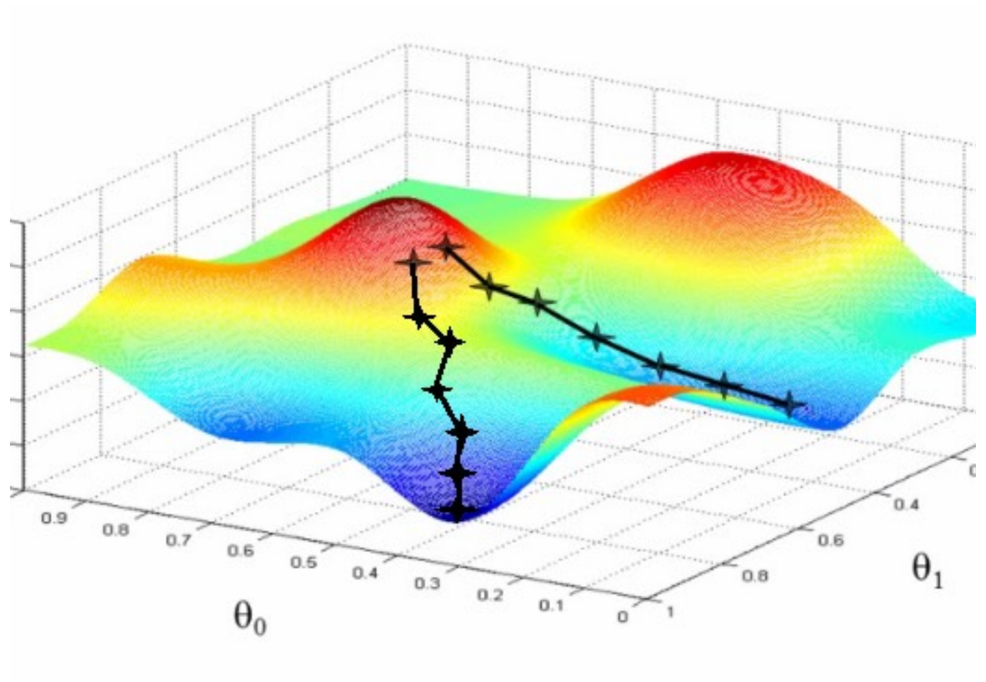
and even $P : \mathrm{real}$ may contain lambdas,
eg $(\lambda f \,.\, \lambda x \,.\, f(x + x))\,(\exp)(2)$

high-dimensional input

program P with many parameters

eg a neural network with many layers, and different weights for the activation functions

low-dimensional output



https://www.youtube.com/watch?v=5u4G23_OohI

ie differentiate the function described by P

aim: optimise the parameters for P

so that, eg, it classifies cats as cats as often as possible

can be done numerically, but it's hard in general!

= a serious bottleneck

can we write an algorithm to calculate derivatives exactly?

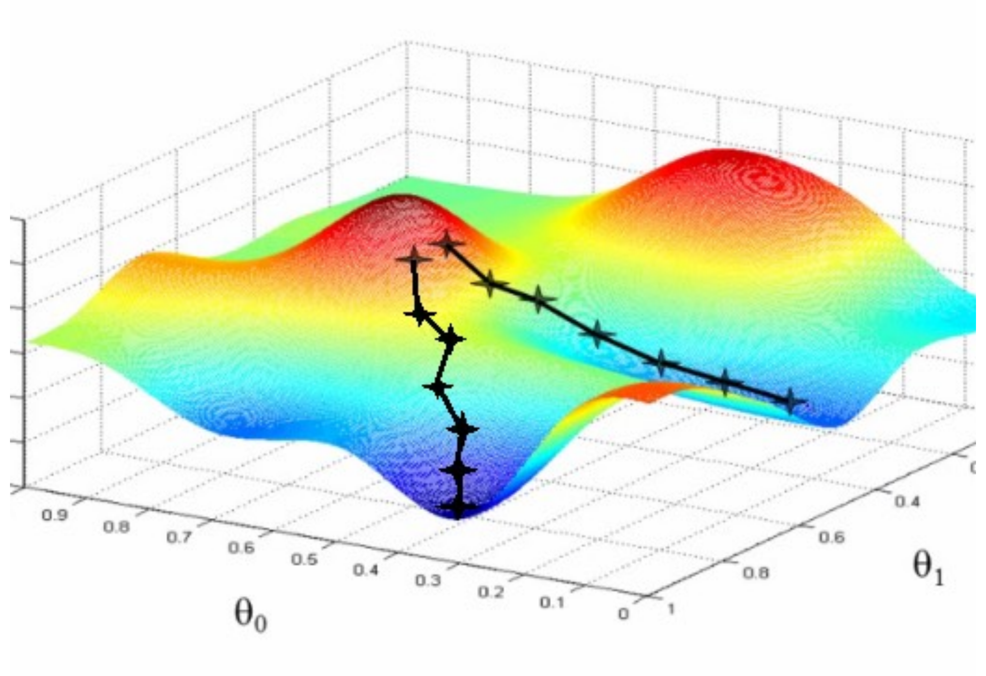
…and can we prove this is correct?

high-dimensional input

program P with many parameters

eg a neural network with many layers, and different weights for the activation functions

low-dimensional output



https://www.youtube.com/watch?v=5u4G23_OohI

ie differentiate the function described by P

aim: optimise the parameters for P

so that, eg, it classifies cats as cats as often as possible

can be done numerically, but it's hard in general!

can we write an algorithm to calculate derivatives exactly?

…and can we prove this is correct?

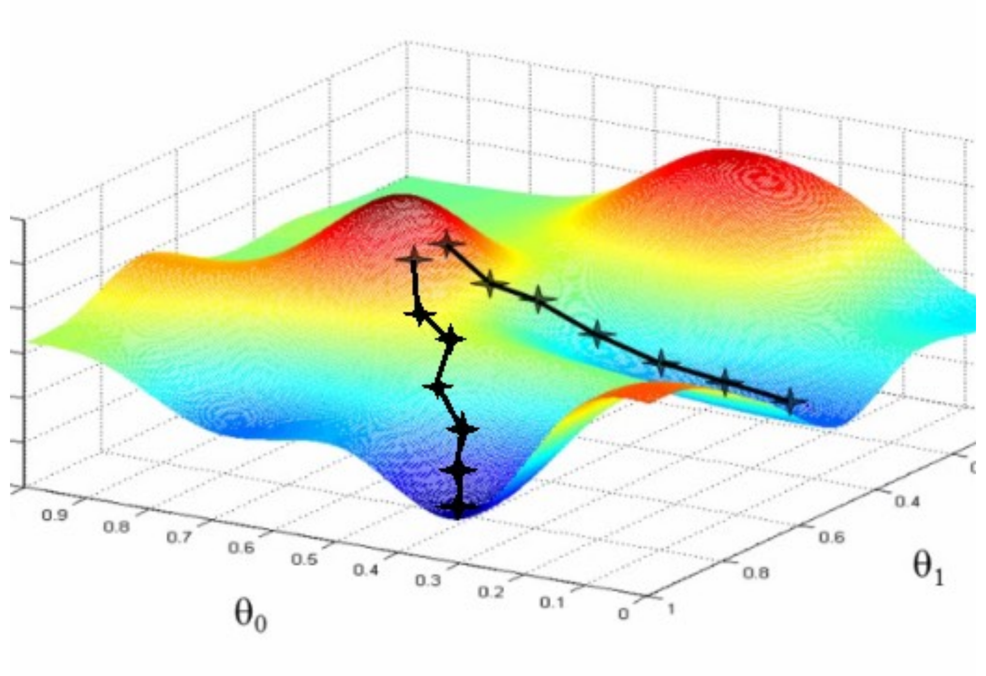
the natural semantic model for studying this problem does not support higher-order functions

high-dimensional input

program P with many parameters

eg a neural network with many layers, and different weights for the activation functions

low-dimensional output



https://www.youtube.com/watch?v=5u4G23_OohI

ie differentiate the function described by P

aim: optimise the parameters for P

so that, eg, it classifies cats as cats as often as possible

can be done numerically, but it's hard in general!

can we write an algorithm to calculate derivatives exactly?

…and can we prove this is correct?

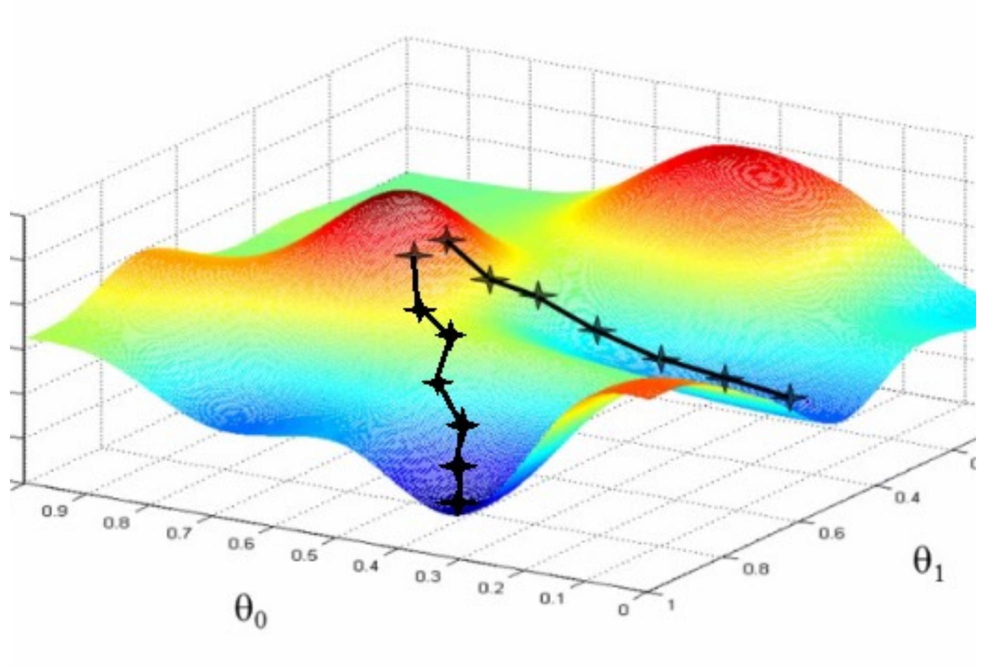
the natural semantic model for studying this problem does not support higher-order functions

high-dimensional input

program P with many parameters

eg a neural network with many layers, and different weights for the activation functions

low-dimensional output



https://www.youtube.com/watch?v=5u4G23_OohI

ie differentiate the function described by P

aim: optimise the parameters for P

so that, eg, it classifies cats as cats as often as possible

can be done numerically, but it's hard in general!

can we write an algorithm to calculate derivatives exactly?

…and can we prove this is correct?

the natural semantic model for studying this problem does not support higher-order functions

we need a CCC that supports some notion of derivative

The category of diffeological spaces

Diff is a nice semantic model! It has:

(1) cartesian closure = can model product and function types

(2) a full embedding $\mathrm{CartSp} \to \mathrm{Diff}$ = conservativity over the natural model, good ways to interpret reals etc

(3) coproducts = can interpret sum types (~ disjoint unions)

(4) initial algebras for endofunctors = can interpret lists and similar inductive types

Proving correctness of automatic differentiation

[Huot, Staton, Vakar]

The strategy:

(a) interpret programs P in Diff

(b) prove that $[\![P : \mathrm{real}]\!]$ always lands in $\mathrm{Cart}Sp$, even if it has lambdas

(c) prove a correctness property for differentiation, at every type

(d) deduce correctness of the $D(-)$ algorithm at type real

Proving correctness of automatic differentiation

[Huot, Staton, Vakar]

The strategy:

(a) interpret programs P in Diff

(b) prove that $[[P : \mathrm{real}]]$ always lands in $\mathrm{Cart}Sp$, even if it has lambdas

(c) prove a correctness property for differentiation, at every type

(d) deduce correctness of the $D(-)$ algorithm at type real

Proving correctness of automatic differentiation

[Huot, Staton, Vakar]

so there's a good
notion of derivative
for these programs

The strategy:

(a) interpret programs P in Diff

(b) prove that $[\![P : \mathrm{real}]\!]$ always lands in CartSp, even if it has lambdas

(c) prove a correctness property for differentiation, at every type

(d) deduce correctness of the $D(-)$ algorithm at type real

Proving correctness of automatic differentiation

[Huot, Staton, Vakar]

so there's a good notion of derivative for these programs

The strategy:

(a) interpret programs P in Diff

(b) prove that $[\![P : \mathrm{real}]\!]$ always lands in CartSp, even if it has lambdas

(c) prove a correctness property for differentiation, at every type

(d) deduce correctness of the $D(-)$ algorithm at type real

need this to handle open variables

Proving correctness of automatic differentiation

[Huot, Staton, Vakar]

so there's a good
notion of derivative
for these programs

The strategy:

(a) interpret programs P in Diff

(b) prove that $[\![P : \mathrm{real}]\!]$ always lands in CartSp, even if it has lambdas

(c) prove a correctness property for differentiation, at every type

(d) deduce correctness of the $D(-)$ algorithm at type real

need this to handle
open variables

Why do denotational semanticists care about Diff?

It provides a good semantic model for differentiable functional programming

…including function types

…which is conservative over the natural model in CartSp

Why do denotational semanticists care about Diff?

It provides a good semantic model for differentiable functional programming

…including function types

…which is conservative over the natural model in CartSp

So we can prove facts about derivatives of programs,

…including higher-order ones

…and thereby verify automatic differentiation algorithms

Why do denotational semanticists care about Diff?

It provides a good semantic model for differentiable functional programming

…including function types

…which is conservative over the natural model in CartSp

So we can prove facts about derivatives of programs,

…including higher-order ones

…and thereby verify automatic differentiation algorithms

And, at type real the interpretation coincides with the natural one

Diff **at work for semantics**

(1) An analogy
(2) Adding recursion
(3) Cutting down the model: full abstraction

Probabilistic programming

Idea:

(1) programs express statistical models,
including conditioning on observations

(2) return the corresponding distribution (often via sampling algorithms)

Probabilistic programming

Idea:

(1) programs express statistical models,
including conditioning on observations

(2) return the corresponding distribution (often via sampling algorithms)

```
normalise(
  let x = sample(bernoulli(0.8));
  let r = (if x then 10 else 3);
  observe 0.45 from exponential(r)
  return(x)
)
```

How do we interpret probabilistic programs?

What is a good semantic model?

Probabilistic programming

Idea:

(1) programs express statistical models,
including conditioning on observations

(2) return the corresponding distribution (often via sampling algorithms)

```
normalise(
  let x = sample(bernoulli(0.8));
  let r = (if x then 10 else 3);
  observe 0.45 from exponential(r)
  return(x)
)
```

probabilistic programs ‘should’ be interpreted by measurable functions

but $\mathbf{Meas}$ is not cartesian closed!

= no way to interpret higher-order functions

How do we interpret probabilistic programs?

What is a good semantic model?

Quasi-Borel spaces [Heunen, Kammar, Moss, Scibior, Staton, Vakar, Yang]

Diff = category of concrete sheaves on cartesian manifolds

QBS = category of concrete sheaves on standard Borel spaces

always a quasi-topos, in particular a CCC

QBS provides a good semantic model for probabilistic programming,
just as Diff provides a good semantic model for differentiable programming

Diff at work for semantics

(1) An analogy: quasi-Borel spaces [Heunen, Kammar, Moss, Scibior, Staton, Vakar, Yang]
(2) Adding recursion
(3) Cutting down the model: full abstraction

Adding recursion to simply-typed λ-calculus

[Scott, Plotkin,…]

$\mathrm{plus} : \mathrm{N} \times \mathrm{N} \to \mathrm{N}$ is the least map satisfying

$$\mathrm{plus}(x, 0) = x$$

$$\mathrm{plus}(x, y + 1) = \mathrm{plus}(x, y) + 1$$

Adding recursion to simply-typed λ-calculus

[Scott, Plotkin,…]

$\mathrm{plus} : \mathrm{N} \times \mathrm{N} \to \mathrm{N}$ is the least map satisfying

$$\mathrm{plus}(x, 0) = x$$
$$\mathrm{plus}(x, y+1) = \mathrm{plus}(x, y) + 1$$

recursion in simply-typed λ-calculus:

$$\frac{P : A \to A}{\mathrm{fix}(M) : A} \qquad \mathrm{fix}(M) \rightsquigarrow M\,\big(\mathrm{fix}(M)\big)$$

Adding recursion to simply-typed λ-calculus

[Scott, Plotkin,…]

$\mathrm{plus} : \mathrm{N} \times \mathrm{N} \to \mathrm{N}$ is the least map satisfying

$$\mathrm{plus}(x,0) = x$$
$$\mathrm{plus}(x, y+1) = \mathrm{plus}(x, y) + 1$$

recursion in simply-typed λ-calculus:

$$\frac{P : A \to A}{\mathrm{fix}(M) : A} \qquad\qquad \mathrm{fix}(M) \rightsquigarrow M\left(\mathrm{fix}(M)\right)$$

$$\mathrm{fix}(M) =_{\beta\eta} M\left(\mathrm{fix}(M)\right)$$

Adding recursion to simply-typed λ-calculus

[Scott, Plotkin,...]

$\mathrm{plus} : \mathrm{N} \times \mathrm{N} \to \mathrm{N}$ is the least map satisfying

$$\mathrm{plus}(x,0) = x$$
$$\mathrm{plus}(x, y+1) = \mathrm{plus}(x,y) + 1$$

$$\frac{P : A \to A}{\mathrm{fix}(M) : A}$$

$$\mathrm{fix}(M) \rightsquigarrow M\left(\mathrm{fix}(M)\right)$$

$$\mathrm{fix}(M) =_{\beta\eta} M\left(\mathrm{fix}(M)\right)$$

Adding recursion to simply-typed λ-calculus

[Scott, Plotkin,…]

$\mathrm{plus} : \mathrm{N} \times \mathrm{N} \to \mathrm{N}$ is the least map satisfying

$$\mathrm{plus}(x, 0) = x$$

$$\mathrm{plus}(x, y + 1) = \mathrm{plus}(x, y) + 1$$

$$\frac{P : A \to A}{\mathrm{fix}(M) : A}$$

$$\mathrm{fix}(M) \rightsquigarrow M\left(\mathrm{fix}(M)\right)$$

$$\mathrm{plus} := \mathrm{fix}\big(\overbrace{\lambda p \,.\, \lambda x \,.\, \lambda y \,.\, \text{if } y == 0 \text{ then } x \text{ else } p\, x\, (y - 1)}^{M}\big)$$

Adding recursion to simply-typed λ-calculus

[Scott, Plotkin,…]

plus : $\mathrm{N} \times \mathrm{N} \to \mathrm{N}$ is the least map satisfying

$$\mathrm{plus}(x,0) = x$$

$$\mathrm{plus}(x, y+1) = \mathrm{plus}(x, y) + 1$$

$$\frac{P : A \to A}{\mathrm{fix}(M) : A} \qquad \mathrm{fix}(M) \rightsquigarrow M\left(\mathrm{fix}(M)\right)$$

$$\mathrm{plus} := \mathrm{fix}\Big(\overbrace{\lambda p\,.\,\lambda x\,.\,\lambda y\,.\,\text{if } y == 0 \text{ then } x \text{ else } p\,x\,(y-1)}^{M}\Big)$$

$$\begin{aligned} \mathrm{plus} &\rightsquigarrow M\,(\mathrm{plus}) \\ &\rightsquigarrow_\beta \lambda x\,.\,\lambda y\,.\,\text{if } y == 0 \text{ text } x \text{ else plus}\,x\,(y-1) \end{aligned}$$

Adding recursion to simply-typed λ-calculus

[Scott, Plotkin,…]

$\mathrm{plus} : \mathrm{N} \times \mathrm{N} \to \mathrm{N}$ is the least map satisfying

$$\mathrm{plus}(x,0) = x$$

$$\mathrm{plus}(x, y+1) = \mathrm{plus}(x, y) + 1$$

$$\frac{P : A \to A}{\mathrm{fix}(M) : A} \qquad \mathrm{fix}(M) \rightsquigarrow M\,\big(\mathrm{fix}(M)\big)$$

$$\mathrm{plus} := \mathrm{fix}\big(\overbrace{\lambda p\,.\,\lambda x\,.\,\lambda y\,.\,\text{if } y == 0 \text{ then } x \text{ else } p\,x\,(y-1)}^{M}\big)$$

$$\mathrm{plus} \rightsquigarrow M\,(\mathrm{plus})$$

$$\rightsquigarrow_\beta \lambda x\,.\,\lambda y\,.\,\text{if } y == 0 \text{ text } x \text{ else plus } x\,(y-1)$$

$$\mathrm{plus}\,x\,y \rightsquigarrow \big(\lambda x\,.\,\lambda y\,.\,\text{if } y == 0 \text{ then } x \text{ else } \mathrm{plus}\,x\,(y-1)\big)\; x\,y$$

$$\rightsquigarrow \text{if } y == 0 \text{ then } x \text{ else } \mathrm{plus}\,x\,(y-1)$$

Adding recursion to the semantics [Scott, Plotkin,…]

$$\frac{P : A \to A}{\mathrm{fix}(M) : A} \qquad \mathrm{fix}(M) \rightsquigarrow M\,\big(\mathrm{fix}(M)\big)$$

$$\mathrm{fix}(M) =_{\beta\eta} M\,\big(\mathrm{fix}(M)\big)$$

Adding recursion to the semantics [Scott, Plotkin,…]

$$\frac{P : A \to A}{\mathrm{fix}(M) : A} \qquad \mathrm{fix}(M) \rightsquigarrow M\ \big(\mathrm{fix}(M)\big)$$

$$\mathrm{fix}(M) =_{\beta\eta} M\ \big(\mathrm{fix}(M)\big)$$

Standard semantics = ω-complete partial orders with a bottom element

Adding recursion to the semantics [Scott, Plotkin,…]

$$\frac{P : A \to A}{\mathrm{fix}(M) : A} \qquad \mathrm{fix}(M) \rightsquigarrow M\left(\mathrm{fix}(M)\right)$$

$$\mathrm{fix}(M) =_{\beta\eta} M\left(\mathrm{fix}(M)\right)$$

Standard semantics = ω-complete partial orders with a bottom element

A Scott domain is a partially ordered set $(X, \leq, \perp)$ where

(1) every chain $x_0 \leq x_1 \leq \cdots \leq x_n \leq \cdots$ has a least upper bound

(2) $\perp \leq x$ for all x

Adding recursion to the semantics [Scott, Plotkin,…]

$$\frac{P : A \to A}{\mathrm{fix}(M) : A} \qquad \mathrm{fix}(M) \rightsquigarrow M\left(\mathrm{fix}(M)\right)$$

$$\boxed{\mathrm{fix}(M) =_{\beta\eta} M\left(\mathrm{fix}(M)\right)}$$

Standard semantics = ω-complete partial orders with a bottom element

A Scott domain is a partially ordered set $(X, \leq, \perp)$ where

(1) every chain $x_0 \leq x_1 \leq \cdots \leq x_n \leq \cdots$ has a least upper bound

(2) $\perp \leq x$ for all x

by Tarski's fixpoint theorem

Forms a CCC, and every map has a least fixed point

x such that $f(x) = x$

$[\![\mathrm{fix}(M)]\!] =$ least fixed point of $[\![M]\!]$

Also an analogous construction for quasi-Borel spaces!

[Vakar, Kammar, Staton]

Adding recursion to $\mathbf{Diff}$ [Vakar]

A Scott domain is a partially ordered set $(X, \leq, \perp)$ where

(1) every chain $x_0 \leq x_1 \leq \cdots \leq x_n \leq \cdots$ has a least upper bound

(2) $\perp \leq x$ for all x

Forms a CCC, and every map has a least fixed point

x such that $f(x) = x$

$[\![\mathrm{fix}(M)]\!] =$ least fixed point of $[\![M]\!]$

Also an analogous construction for quasi-Borel spaces!

[Vakar, Kammar, Staton]

Adding recursion to Diff [Vakar]

A Scott domain is a partially ordered set $(X, \leq, \perp)$ where

(1) every chain $x_0 \leq x_1 \leq \cdots \leq x_n \leq \cdots$ has a least upper bound

(2) $\perp \leq x$ for all x

Forms a CCC, and every map has a least fixed point

x such that $f(x) = x$

$[\![\mathrm{fix}(M)]\!] =$ least fixed point of $[\![M]\!]$

def: an ω-diffeological space $(X, \mathscr{P}_X, \leq)$ is

a diffeological space $(X, \mathscr{P}_X)$

…such that $(X, \leq)$ is a domain

…and $\mathscr{P}_X^U$ is closed under least upper bounds of chains

Also an analogous construction for quasi-Borel spaces!

[Vakar, Kammar, Staton]

Adding recursion to Diff [Vakar]

A Scott domain is a partially ordered set $(X, \leq, \bot)$ where

(1) every chain $x_0 \leq x_1 \leq \cdots \leq x_n \leq \cdots$ has a least upper bound

(2) $\bot \leq x$ for all x

Forms a CCC, and every map has a least fixed point

x such that $f(x) = x$

$[\![\mathrm{fix}(M)]\!] =$ least fixed point of $[\![M]\!]$

def: an ω-diffeological space $(X, \mathscr{P}_X, \leq)$ is

a diffeological space $(X, \mathscr{P}_X)$

…such that $(X, \leq)$ is a domain

…and $\mathscr{P}_X^U$ is closed under least upper bounds of chains

Can extend correctness results for AD to languages with recursion!

Diff **at work for semantics**

(1) An analogy: quasi-Borel spaces [Heunen, Kammar, Moss, Scibior, Staton, Vakar, Yang]

(2) Adding recursion [Vakar] [Vakar, Kammar, Staton]

(3) Cutting down the model: full abstraction

Cutting down Diff [Kammar, Katsumata, S.]

Given $P \simeq_{\text{ctx}} Q$, can we deduce $[\![P]\!] = [\![Q]\!]$?

is the model fully abstract?

Cutting down Diff [Kammar, Katsumata, S.]

Given $P \simeq_{\mathrm{ctx}} Q$, can we deduce $[\![P]\!] = [\![Q]\!]$?

is the model fully abstract?

In general, no!

$[\![P]\!]$ and $[\![Q]\!]$ can agree on all definable things, but still differ!

the semantics expresses richer behaviour than the syntax

Cutting down Diff [Kammar, Katsumata, S.]

Given $P \simeq_{\mathrm{ctx}} Q$, can we deduce $[\![P]\!] = [\![Q]\!]$?

is the model fully abstract?

In general, no!

$[\![P]\!]$ and $[\![Q]\!]$ can agree on all definable things, but still differ!

the semantics expresses richer behaviour than the syntax

Solution:

refine the model so every $f : [\![A]\!] \to [\![B]\!]$ is definable

difficult bit: doing this for exponentials

Cutting down Diff [Kammar, Katsumata, S.]

Given $P \simeq_{\mathrm{ctx}} Q$, we can't deduce $[\![P]\!] = [\![Q]\!]$

is the model fully abstract?

Solution:

refine the model so every $f : [\![A]\!] \to [\![B]\!]$ is definable

difficult bit: doing this for exponentials

idea: internalise the idea that f is definable
if it preserves the property of being definable

Cutting down Diff [Kammar, Katsumata, S.]

Given $P \simeq_{\mathrm{ctx}} Q$, we can't deduce $[\![P]\!] = [\![Q]\!]$

$\boxed{\text{is the model fully abstract?}}$

Solution:

refine the model so every $f : [\![A]\!] \to [\![B]\!]$ is definable

difficult bit: doing this for exponentials

idea: internalise the idea that f is definable
if it preserves the property of being definable

objects = diffeological spaces paired with a family of relations
morphisms = smooth maps preserving the relations

$$\text{(new model)} \xrightarrow{\quad} \mathrm{Diff}$$

(new model)

$\downarrow$ preserves primitives and products,
but not exponentials

Diff

choose the class of relations intensionally
so maps preserving the relation are definable

Diff at work for semantics

(1) An analogy: quasi-Borel spaces [Heunen, Kammar, Moss, Scibior, Staton, Vakar, Yang]

(2) Adding recursion [Vakar] [Vakar, Kammar, Staton]

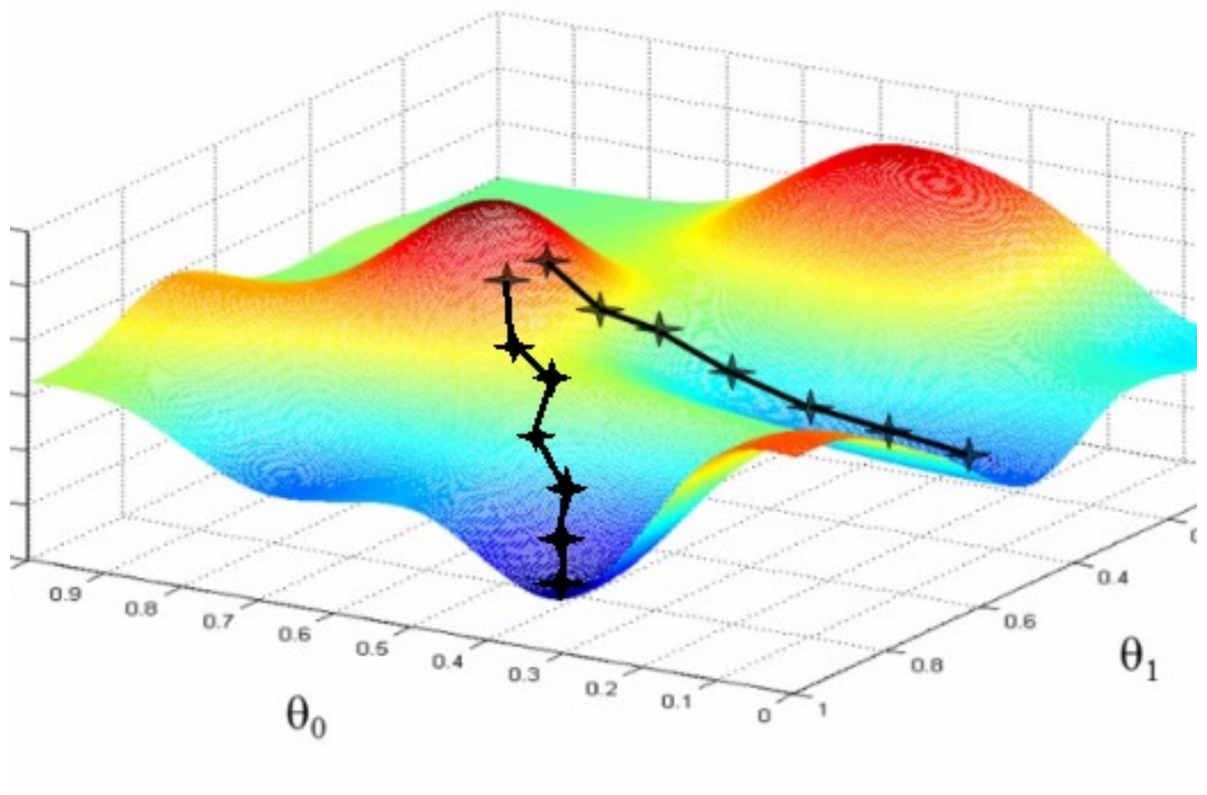
(3) Cutting down the model: full abstraction [Kammar, Katsumata, S.]

Denotational semantics:

- idealised functional programming language
  = simply-typed λ-calculus (+ extensions)
- interchangeability of programs
  = observational equivalence finer than equality-on-arguments!
- interpret programs in CCCs (+ extensions)

$$P \mapsto D(P)$$

Diff is a good model for studying
automatic differentiation of programs



https://www.youtube.com/watch?v=5u4G23_OohI