Lambda calculus and practical ML

CSCI3180 Principles of Programming Languages

Spring 2018

Sunny Lai

Lambda calculus and practical ML

- Lambda calculus and SML
  - Currying
  - Reduction Strategies

- Practical ML
  - OCaml
  - OCaml vs SML
  - 99 Problems (solved) in Ocaml
  - ReasonML

Currying

- In mathematics and computer science,
- **currying** is the technique of translating the evaluation of a function that takes multiple arguments (or a tuple of arguments) into evaluating a sequence of functions, each with a single argument.
- Currying is related to, but not the same as, partial application.

https://en.wikipedia.org/wiki/Currying

Currying

- Last tutorial we define the add function

Tuple

- `fun add(x,y):int = x+y;`
- `val add = fn : int * int -> int;`

Tuple

- Takes in a tuple with two integer (type inference), and return an integer
- What if we want to use 'Partial application'?
  - i.e. partially instantiated or specialized functions where some (but not all) arguments are supplied

Currying

- Recall the example in lecture
- $define\ Curry\ = \lambda f\,.\lambda x\,.\lambda y\,.f < x, y >$
- $define\ Uncurry\ =$
  $\lambda f\,.\lambda p\,.f\ (head\ p)(tail\ p)$

- How can we use them in ML?

Currying in JavaScript

```
function add (a) {
  return function (b) {
    return a + b;
  }
}
```

Currying

- Last tutorial we define the add function

Uncurried

Tuple

- `fun add(x,y):int = x+y;`
- `val add = fn : int * int -> int;`

Tuple

Curried

- `fun add x y :int = x+y;`
- `val add = fn : int -> int -> int;`

Integer argument

Integer argument

Currying

- `fun curry f x y = f(x, y);`
  - `val curry =`
    `fn : ('a * 'b -> 'c) -> 'a -> 'b -> 'c`
- `fun uncurry g (x, y) = g x y;`
  - `val uncurry =`
    `fn : ('a -> 'b -> 'c) -> 'a * 'b -> 'c`

https://stackoverflow.com/questions/8395564/what-are-curry-and-uncurry-in-high-order-functions-in-ml
https://stackoverflow.com/questions/37006575/i-want-to-prove-uncurrycurryf-f-curryuncurryg-g-with-using-ml

Currying

Tuple

- fun foo (a,x) = a*x+10

  Uncurried

  - val foo = fn : int * int -> int

Int Int

- fun bar a x = a*x+20

  Curried

  - val bar = fn : int -> int -> int

- curry foo;
  - val it = fn : int -> int -> int
- uncurry bar;
  - val it = fn : int * int -> int

https://stackoverflow.com/questions/8395564/what-are-curry-and-uncurry-in-high-order-functions-in-ml
https://stackoverflow.com/questions/37006575/i-want-to-prove-uncurrycurryf-f-curryuncurryg-g-with-using-ml

Currying

- Curried functions are useful because they allow us to create partially instantiated or specialized functions where some (but not all) arguments are supplied.

- Particularly useful for high-order functions
- ```
  fun twice (f: int->int) =
      fn(x: int) => f (f (x))
  ```

  `(int->int)->int->int.`

- ```
  fun twice (f: int->int) (x:int) : int
  = f (f (x))
  ```

  syntactic sugar

http://www.cs.cornell.edu/courses/cs312/2008sp/recitations/rec03.html

Currying in lambda calculus

- Recall the example in lecture

Functions can be named

define $Twice = \lambda f\,.\,\lambda x\,.\,f\ (f\ x)$

then

$(Twice\ (\lambda n\,.\,(add\ n\ 1))\ 5) = 7$

$((\lambda f\,.\,\lambda x\,.\,f\ (f\ x))\ (\lambda n\,.\,(add\ n\ 1))\ 5) = 7$

But in what ordered are they evaluated?

Reduction Strategies

- Applicative order reduction:
  - Always fully evaluate the arguments of a function before evaluating the function itself (leftmost innermost β- redex)

- Normal order reduction:
  - The expression would be reduced from the outside in (leftmost outermost β- redex)

https://cs.stackexchange.com/questions/7702/applicative-order-and-normal-order-in-lambda-calculus https://cs.stackexchange.com/questions/54000/is-applicative-order-and-normal-order-evaluation-models-definition-contradictor/54004

Reduction Strategies

Let's look at `(square (tracing 2))` in this model. In the notation I've presented, that's $((\lambda x.\,(*\,x\,x))(\mathtt{tracing}\,2))$. In normal order:

$$
\begin{aligned}
((\lambda x.\,(*\,x\,x))\,(\mathtt{tracing}\,2)) &\longrightarrow (*\,(\mathtt{tracing}\,2)\,(\mathtt{tracing}\,2)) && \text{by } (\beta) \\
&\xrightarrow{2} (*\,2\,(\mathtt{tracing}\,2)) && \text{by } (\delta_{\mathtt{tracing}}) \\
&\xrightarrow{2} (*\,2\,2) && \text{by } (\delta_{\mathtt{tracing}}) \\
&\longrightarrow 4 && \text{by } (\delta_{*})
\end{aligned}
$$

In applicative order:

$$
\begin{aligned}
((\lambda x.\,(*\,x\,x))\,(\mathtt{tracing}\,2)) &\xrightarrow{2} ((\lambda x.\,(*\,x\,x))\,2) && \text{by } (\delta_{\mathtt{tracing}}) \\
&\longrightarrow (*\,2\,2) && \text{by } (\beta) \\
&\longrightarrow 4 && \text{by } (\delta_{*})
\end{aligned}
$$

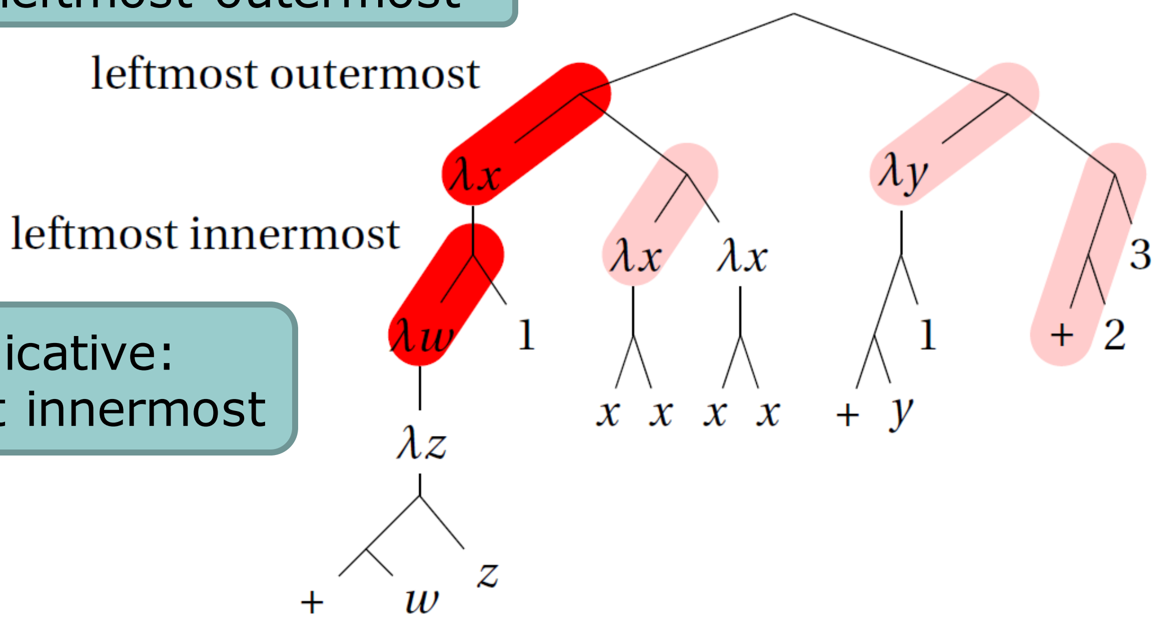
(I used left-to-right evaluation for function arguments; specifying this is necessary to get a fully deterministic evaluation order.) This shows how $((\lambda x.\,(*\,x\,x))\,(\mathtt{tracing}\,2))$ evaluates to 4 with the trace $2, 2$ under normal order, but to 4 with the trace 2 under applicative order.

Reduction Strategies

$$\Big(\big(\lambda x\,.\,((\lambda w\,.\,\lambda z\,.\,+\;w\;z)\;1)\big)\;\big((\lambda x\,.\,x\;x)\;(\lambda x\,.\,x\;x)\big)\Big)\;\big((\lambda y\,.\,+\;y\;1)\;(+\;2\;3)\big)$$

Normal: leftmost outermost

Applicative: leftmost innermost



Reduction Strategies

- $(\lambda x. x^2(\lambda x. (x+1) \ 2)))$
$\rightarrow \left(\lambda x. x^2(2+1)\right)$

Applicative: leftmost innermost

$\rightarrow \ \left(\lambda x. x^2(3)\right)$
$\rightarrow \ 3^2 \ \rightarrow \ 9$

- $\left(\lambda x. x^2 \left(\lambda x. (x+1) \ 2\right)\right)$
$\rightarrow \ \left(\lambda x. (x+1) \ \ 2\right)^2$

Normal: leftmost outermost

$\rightarrow \ (2+1)^2 \ \rightarrow 3^2 \ \rightarrow \ 9$

https://cs.stackexchange.com/questions/7702/applicative-order-and-normal-order-in-lambda-calculus

Reduction Strategies

- Different reduction strategies leads to:
  - different final result?
  - different intermediate result?

Practical ML (OCaml)

OCaml

- Trading
- Static type checking(JS)
- System operations
- Code analysis

Companies using OCaml

> *"OCaml helps us to quickly adapt to changing market conditions, and go from prototypes to production systems with less effort ... Billions of dollars of transactions flow through our systems every day, so getting it right matters."* — Jane Street

Facebook, United States

Facebook has built a number of major development tools using OCaml. Hack is a compiler for a variant of PHP that aims to reconcile the fast development cycle of PHP with the discipline provided by static typing. Flow is a similar project that provides static type checking for Javascript. Both systems are highly responsive, parallel programs that can incorporate source code changes in real time. Pfff is a set of tools for code analysis, visualizations, and style-preserving source transformations, written in OCaml, but supporting many languages.

Docker, United States

Docker provides an integrated technology suite that enables development and IT operations teams to build, ship, and run distributed applications anywhere. Their native applications for Mac and Windows, use OCaml code taken from the MirageOS library operating system project.

Bloomberg L.P., United States

Bloomberg, the global business and financial information and news leader, gives influential decision makers a critical edge by connecting them to a dynamic network of information, people and ideas. Bloomberg employs OCaml in a advanced financial derivatives risk management application delivered through its Bloomberg Professional service.

Citrix, United Kingdom

Citrix uses OCaml in XenServer, a world-class server virtualization system. We also offer a full open-source variant of XenServer called the Xen Cloud Platform, or XCP. Follow along with our OCaml development at github.com/xen-org. This work was originally presented by Anil Madhavapeddy at CUFP 2008. See his abstract and

Jane Street, United States

Jane Street is a quantitative proprietary trading firm that operates around the clock and around the globe. They bring a deep understanding of markets, a scientific approach, and innovative technology to bear on the problem of trading profitably in the world's highly competitive financial markets. Jane Street is perhaps the largest commercial user of OCaml, and has attracted a very strong team of functional programmers. They use OCaml for everything, from research infrastructure to trading systems to operations and accounting systems. Jane Street has over 50 OCaml programmers and over a million lines of OCaml, powering a technology platform that trades billions of dollars every day. See the GitHub page for their open source software.

https://ocaml.org/learn/companies.html

OCaml vs SML

- Comparing Objective Caml and Standard ML
  - http://adam.chlipala.net/mlcomp/
- Standard ML and Objective Caml, Side by Side
  - https://people.mpi-sws.org/~rossberg/sml-vs-ocaml.html

99 Problems (solved) in OCaml

- 99 Problems (solved) in OCaml
  - https://ocaml.org/learn/tutorials/99problems.html

99 Problems (solved) in OCaml

This section is inspired by Ninety-Nine Lisp Problems which in turn was based on "Prolog problem list". For each of these questions, some simple tests are shown—they may also serve to make the question clearer if needed. To work on these problems, we recommend you first install OCaml or use it inside your browser. The source of the following problems is available on GitHub.

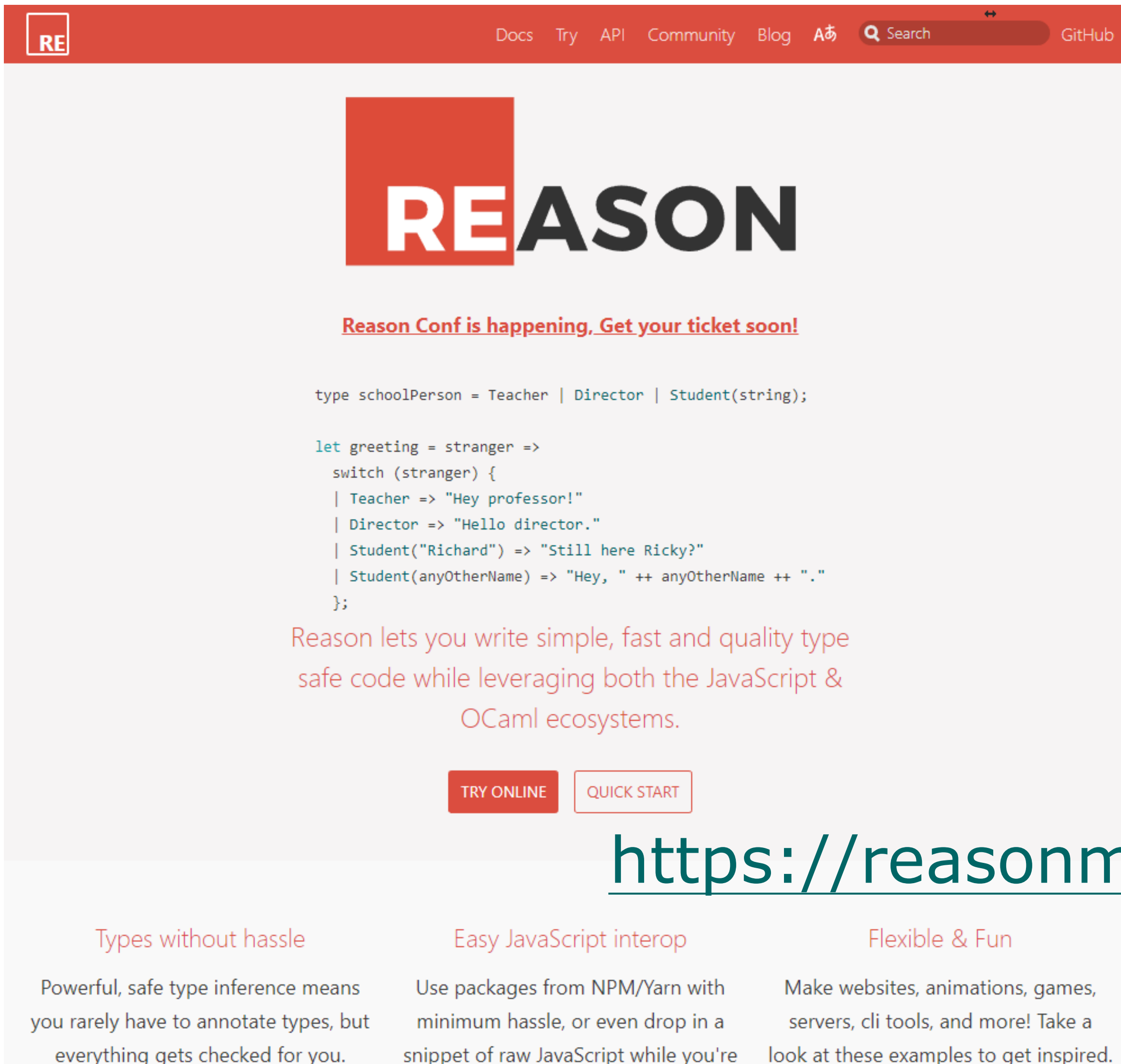
Working with lists

1. Write a function `last : 'a list -> 'a option` that returns the last element of a list. (*easy*)

Solution

```
# last [ "a" ; "b" ; "c" ; "d" ];;
- : string option = Some "d"
# last [];;
```

Reason(reasonml)



https://reasonml.github.io/

RE Docs Try API Community Blog A Search GitHub

Pattern Matching!

EDIT

Make sure you've read on Variant first.

We're finally here! Pattern matching is one of *the* best features of the language. It's like destructuring, but comes with even more help from the type system.

Usage

Consider a variant:

```
type payload =
  | BadResult(int)
  | GoodResult(string)
  | NoResult;
```

While using the `switch` expression on it, you can "destructure" it:

```
let data = GoodResult("Product shipped!");

let message =
  switch (data) {
  | GoodResult(theMessage) => "Success! " ++ theMessage
  | BadResult(errorCode) => "Something's wrong. The error code is: " ++ string_of_ir
  };
```

Notice how we've destructured `data` while handling each different case. The above `switch` will give you a compiler warning:

```
Warning 8: this pattern-matching is not exhaustive.
```

- Pattern Matching
  - https://reasonml.github.io/docs/en/pattern-matching.html

Next tutorial

- Prolog
- Assignment 4