A Mathematical Model of OCaml

Speaker: David Walker
COS 326
Princeton University

From Code to Abstract Specification

OCaml code can give a language semantics

- advantage: it can be executed, so we can try it out
- advantage: it is amazingly concise
  - especially compared to what you would have written in Java
- disadvantage: it is a little ugly to operate over concrete ML datatypes like "Op(e1,Plus,e2)" as opposed to "e1 + e2"

From Code to Abstract Specification

PL has a notation for these specifications:

- it has a mathematical “feel” that makes PL researchers feel special and gives us *goosebumps* inside

- it operates over abstract expression syntax like “e1 + e2”

- it is useful to know this notation if you want to read specifications of programming language semantics
  - e.g.: Standard ML (of which OCaml is a descendent) has a formal definition given in this notation (and C, and Java; but not OCaml…)
  - e.g.: most papers in the conference POPL (ACM Principles of Prog. Lang.)

Goal

Our goal is to explain how an expression e evaluates to a value v.

Ie, we will define a *relation* between expressions and values.

Formal Inference Rules

We will define the “evaluates to” relation using a set of (inductive) rules that allow us to *prove* that a particular (expression, value) pair is part of the relation.

A rule looks like this:

$$\frac{\text{premise 1} \qquad \text{premise 2} \qquad \ldots \qquad \text{premise 3}}{\text{conclusion}}$$

You read a rule like this:

- “if premise 1 can be proven and premise 2 can be proven and ... and premise n can be proven then conclusion can be proven”

Some rules have no premises

- this means their conclusions are always true
- we call such rules “axioms” or “base cases”

An example rule

As a rule:

$$\frac{\text{e1 --> v1} \qquad \text{e2 --> v2} \qquad \text{eval\_op (v1, op, v2) == v'}}{\text{e1 op e2 --> v'}}$$

In English:

> "If e1 evaluates to v1
> and e2 evaluates to v2
> and eval_op (v1, op, v2) is equal to v'
> then
> e1 op e2 evaluates to v'

In code:

```
let rec eval (e:exp) : exp =
  match e with
  | Op_e(e1,op,e2) -> let v1 = eval e1 in
                      let v2 = eval e2 in
                      let v' = eval_op v1 op v2 in
                      v'
```

An example rule

As a rule:

$$\frac{i \in Z}{i \;\text{-->}\; i}$$

asserts i is an integer

In English:

> "If the expression is an integer value, it evaluates to itself."

In code:

```
let rec eval (e:exp) : exp =
  match e with
 | Int_e i -> Int_e i
 ...
```

An example rule concerning evaluation

As a rule:

$$\frac{\text{e1 --> v1} \qquad \text{e2 [v1/x] --> v2}}{\text{let x = e1 in e2 --> v2}}$$

In English:

"If e1 evaluates to v1 (which is a *value*)
and e2 with v1 substituted for x evaluates to v2
then let x=e1 in e2 evaluates to v2."

In code:

```
let rec eval (e:exp) : exp =
  match e with
 | Let_e(x,e1,e2) -> let v1 = eval e1 in
                     eval (substitute v1 x e2)
 ...
```

An example rule concerning evaluation

As a rule:

$$\frac{}{\lambda x.e \;\texttt{-->}\; \lambda x.e}$$

typical "lambda" notation for a function with argument x, body e

In English:

"A function value evaluates to itself."

In code:

```
let rec eval (e:exp) : exp =
  match e with
 ...
 | Fun_e (x,e) -> Fun_e (x,e)
 ...
```

An example rule concerning evaluation

As a rule:

$$\frac{e1 \dashrightarrow \lambda x.e \qquad e2 \dashrightarrow v2 \qquad e[v2/x] \dashrightarrow v}{e1\ e2 \dashrightarrow v}$$

In English:

> "if e1 evaluates to a function with argument x and body e
> and e2 evaluates to a value v2
> and e with v2 substituted for x evaluates to v
> then e1 applied to e2 evaluates to v"

In code:

```
let rec eval (e:exp) : exp =
  match e with
 ..
| Call_e (e1,e2) ->
          (match eval e1 with
           | Fun_e (x,e) -> eval (substitute (eval e2) x e)
           | ...)
...
```

An example rule concerning evaluation

As a rule:

$$\frac{\text{e1} \dashrightarrow \text{rec f x = e} \qquad \text{e2} \dashrightarrow \text{v} \qquad \text{e[rec f x = e/f][v/x]} \dashrightarrow \text{v2}}{\text{e1 e2} \dashrightarrow \text{v2}}$$

In English:

“uggh”

In code:

```
let rec eval (e:exp) : exp =
  match e with
     ...
   | (Rec_e (f,x,e)) as f_val ->
        let v = eval e2 in
        substitute f_val (substitute v x e) g
```

Comparison: Code vs. Rules

complete eval code:

```
let rec eval (e:exp) : exp =
  match e with
  | Int_e i -> Int_e i
  | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)
  | Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)
  | Var_e x -> raise (UnboundVariable x)
  | Fun_e (x,e) -> Fun_e (x,e)
  | FunCall_e (e1,e2) ->
      (match eval e1
       | Fun_e (x,e) -> eval (Let_e (x,e2,e))
       | _ -> raise TypeError)
  | LetRec_e (x,e1,e2) ->
     (Rec_e (f,x,e)) as f_val ->
       let v = eval e2 in
       substitute f_val f (substitute v x e)
```

complete set of rules:

$$\frac{i \in \mathbb{Z}}{i \;\text{-->}\; i}$$

$$\frac{e1 \;\text{-->}\; v1 \qquad e2 \;\text{-->}\; v2 \qquad \text{eval\_op}\,(v1,\, op,\, v2) == v}{e1 \;op\; e2 \;\text{-->}\; v}$$

$$\frac{e1 \;\text{-->}\; v1 \qquad e2\,[v1/x] \;\text{-->}\; v2}{\text{let } x = e1 \text{ in } e2 \;\text{-->}\; v2}$$

$$\frac{}{\lambda x.e \;\text{-->}\; \lambda x.e}$$

$$\frac{e1 \;\text{-->}\; \lambda x.e \qquad e2 \;\text{-->}\; v2 \qquad e[v2/x] \;\text{-->}\; v}{e1\; e2 \;\text{-->}\; v}$$

$$\frac{e1 \;\text{-->}\; \text{rec } f\; x = e \qquad e2 \;\text{-->}\; v2 \qquad e[\text{rec } f\; x = e/f][v2/x] \;\text{-->}\; v3}{e1\; e2 \;\text{-->}\; v3}$$

Almost isomorphic:

- one rule per pattern-matching clause
- recursive call to eval whenever there is a --> premise in a rule
- what's the main difference?

Comparison: Code vs. Rules

complete eval code:

```
let rec eval (e:exp) : exp =
  match e with
   | Int_e i -> Int_e i
   | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)
   | Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)
   | Var_e x -> raise (UnboundVariable x)
   | Fun_e (x,e) -> Fun_e (x,e)
   | FunCall_e (e1,e2) ->
       (match eval e1
        | Fun_e (x,e) -> eval (Let_e (x,e2,e))
        | _ -> raise TypeError)
   | LetRec_e (x,e1,e2) ->
      (Rec_e (f,x,e)) as f_val ->
        let v = eval e2 in
        substitute f_val f (substitute v x e)
```

complete set of rules:

$$\frac{i \in Z}{i \longrightarrow i}$$

$$\frac{e1 \longrightarrow v1 \qquad e2 \longrightarrow v2 \qquad \text{eval\_op}\ (v1,\ op,\ v2) == v}{e1\ op\ e2 \longrightarrow v}$$

$$\frac{e1 \longrightarrow v1 \qquad e2\ [v1/x] \longrightarrow v2}{\text{let}\ x = e1\ \text{in}\ e2 \longrightarrow v2}$$

$$\frac{}{\lambda x.e \longrightarrow \lambda x.e}$$

$$\frac{e1 \longrightarrow \lambda x.e \qquad e2 \longrightarrow v2 \qquad e[v2/x] \longrightarrow v}{e1\ e2 \longrightarrow v}$$

$$\frac{e1 \longrightarrow \text{rec}\ f\ x = e \qquad e2 \longrightarrow v2 \qquad e[\text{rec}\ f\ x = e/f][v2/x] \longrightarrow v3}{e1\ e2 \longrightarrow v3}$$

- There's no formal rule for handling free variables
- No rule for evaluating function calls when a non-function in the caller position
- In general, *no rule when further evaluation is impossible*
  - the rules express the *legal evaluations* and say nothing about what to do in error situations
  - the code handles the error situations by raising exceptions
  - type theorists prove that well-typed programs don't run into undefined cases

This Lecture's Model of Computation

This lecture's model of computation is often called the *substitution model*

It models pure programming features succinctly, but non-trivial changes are required to model more sophisticated constructs:

- I/O, exceptions, mutation, concurrency, …
- we can build models of these things, but they aren't as simple.
- ... even modelling substitution was somewhat tricky

It's useful for reasoning about correctness of algorithms and optimizations

– we can use it to formally prove that, for instance:
  - map f (map g xs) == map (comp f g) xs
  - proof: by induction on the length of the list xs, using the definitions of the substitution model

It is *not* useful for reasoning about execution time or space

– more complex models needed there

This Lecture's Model of Computation

This model of computation is often called the *substitution model*

It models pure programming features succinctly, but non-trivial changes are required to model more sophisticated constructs:

- I/O, exceptions, mutation, concurrency, ...
- we can build models of these things, but they aren't a
- ... even modelling substitution was somewhat tricky

It's useful for reaso algorithms s

– we can use ce:

- map f (
- proof: by t xs, using t e substitution m

You can say that again!
I got it wrong the first
time I tried, in 1932.
Fixed the bug by 1934,
though.

Alonzo Church,
1903-1995
Princeton Professor,
1929-1967

It is *not* useful for reasoning about execution time or sp

– more complex models needed there

Church's mistake

substitute:

```
fun xs -> map (+) xs
```

map is free here – it refers to a library function

for f in:

```
fun ys ->
  let map xs = 0::xs in
  f (map ys)
```

the problem was that the value you substituted in had a *free variable* (map) in it that was *captured*.

and if you don't watch out, you will get:

```
fun ys ->
  let map xs = 0::xs in
  (fun xs -> map (+) xs) (map ys)
```

Church's mistake

substitute:

```
fun xs -> map (+) xs
```

for f in:

```
fun ys ->
  let map xs = 0::xs in
  f (map ys)
```

to do it right, you need to rename some variables:

```
fun ys ->
  let z xs = 0::xs in
  (fun xs -> map (+) xs) (z ys)
```

change "map" to "z" before substituting

Recap

In this lecture, we explored a mathematical specification of OCaml expressions

- we specified the evaluation model using a set of *inference rules*
- these inference rules defined a relation between expressions and values
- we found that values evaluated to themselves
  - values are the results of evaluation
  - integer constants and functions both count as values in this model of execution
- and we found that *substitution* is used to handle constructs that involve variable binding
  - let expressions: "let x = e1 in e2" -- substitute e1's value for x in e2
  - function application: "(fun x -> e2) e1" -- substitute e1's value for x in e2
  - recursive function application: "(rec f x = e1) e2" -- like non-recursive functions, but also substitute recursive function for name of function
- more on this in COS 510

Exercise

Try extending the language and rules for evaluation with:

- booleans (true, false, and, or, not, if)
- pairs (with pair creation and field extraction operations)