A Mathematical Model of OCaml

Speaker: David Walker
COS 326
Princeton University
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”
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.)
Our goal is to explain how an expression $e$ evaluates to a value $v$.

I.e., 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:

\[
\text{premise 1} \quad \text{premise 2} \quad \cdots \quad \text{premise 3} \\
\text{conclusion}
\]

You read a rule like this:

– “if premise 1 can be proven and premise 2 can be proven and \ldots 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”
As a rule:

\[ e_1 \rightarrow v_1 \quad e_2 \rightarrow v_2 \quad \text{eval}_\text{op}(v_1, \text{op}, v_2) = v' \]
\[ e_1 \text{ op } e_2 \rightarrow v' \]

In English:

“If \( e_1 \) evaluates to \( v_1 \)
and \( e_2 \) evaluates to \( v_2 \)
and \( \text{eval}_\text{op}(v_1, \text{op}, v_2) \) is equal to \( v' \)
then
\( e_1 \text{ op } e_2 \) evaluates to \( v' \)

In code:

```ocaml
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'
```

"If \( e_1 \) evaluates to \( v_1 \)
and \( e_2 \) evaluates to \( v_2 \)
and \( \text{eval}_\text{op}(v_1, \text{op}, v_2) \) is equal to \( v' \)
then
\( e_1 \text{ op } e_2 \) evaluates to \( v' \)
An example rule

As a rule:

\[ i \in \mathbb{Z} \quad \Rightarrow \quad i \rightarrow i \]

In English:

“If the expression is an integer value, it evaluates to itself.”

In code:

```ml
let rec eval (e:exp) : exp =
    match e with
    | Int_e i -> Int_e i
    ...
```
An example rule concerning evaluation

As a rule:

\[
\begin{align*}
    e_1 & \rightarrow v_1 & e_2[v_1/x] & \rightarrow v_2 \\
    \text{let } x = e_1 \text{ in } e_2 & \rightarrow v_2
\end{align*}
\]

In English:

“\begin{quote} 
If \( e_1 \) evaluates to \( v_1 \) (which is a \textit{value}) and \( e_2 \) with \( v_1 \) substituted for \( x \) evaluates to \( v_2 \) then \( \text{let } x = e_1 \text{ in } e_2 \) evaluates to \( v_2 \).\end{quote}"

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:

\[ \lambda x. e \rightarrow \lambda x. e \]

In English:

“A function value evaluates to itself.”

In code:

```ml
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:

\[
\begin{align*}
  e_1 \rightarrow \lambda x. e & \quad e_2 \rightarrow v_2 & \quad e[v_2/x] \rightarrow v \\
  e_1 e_2 \rightarrow v
\end{align*}
\]

In English:

“if \( e_1 \) evaluates to a function with argument \( x \) and body \( e \) and \( e_2 \) evaluates to a value \( v_2 \) and \( e \) with \( v_2 \) substituted for \( x \) evaluates to \( v \) then \( e_1 \) applied to \( e_2 \) evaluates to \( v \)”

In code:

```ocaml
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:

\[
\begin{align*}
\text{e1} & \rightarrow \text{rec f x = e} & \text{e2} & \rightarrow \text{v} & \text{e[rec f x = e/f][v/x]} & \rightarrow \text{v2} \\
\text{e1 e2} & \rightarrow \text{v2}
\end{align*}
\]

In English:

“uggh”

In code:

```ocaml
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

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?

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 eval code:

complete set of rules:

\[
\begin{align*}
\forall i \in \mathbb{Z} & \quad i \rightarrow i \\
\text{let rec } & \quad v_1 \rightarrow v_1 \quad v_2 \rightarrow v_2 \quad \text{eval_op } (v_1, \text{op}, v_2) \rightarrow v \\
\text{let } & \quad x = e_1 \text{ in } e_2 \rightarrow v_2 \\
\lambda x.e & \rightarrow \lambda x.e \\
\lambda x.e & \rightarrow v_1 \quad e_2 \rightarrow v_2 \quad \text{e[v2/x]} \rightarrow v \\
\text{let rec } & \quad f x = e \quad e_2 \rightarrow v_2 \quad \text{e[rec f x = e/f][v2/x]} \rightarrow v_3 \\
\end{align*}
\]
Comparison: Code vs. Rules

- 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

### Complete eval code:

```ocaml
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:

```
\[ \begin{align*}
i \in \mathbb{Z} \\
i \mapsto & i
\end{align*} \]
```

```
\[ \begin{align*}
e_1 \mapsto v & \quad e_2 \mapsto v_2 \quad \text{eval}_\text{op} (v_1, \text{op}, v_2) = v \\
e_1 \mapsto v & \quad e_2 [v_1/x] \mapsto v_2 \\
\text{let } x = e_1 \text{ in } e_2 \mapsto v_2
\end{align*} \]
```

```
\[ \begin{align*}
\lambda x.e & \mapsto \lambda x.e \\
e_1 \mapsto \lambda x.e & \quad e_2 \mapsto v_2 \quad e[v_2/x] \mapsto v
\end{align*} \]
```

```
\[ \begin{align*}
e_1 \mapsto \text{rec } f x = e & \quad e_2 \mapsto v_2 \quad e[\text{rec } f x = e/f][v_2/x] \mapsto v_3 \\
e_1 e_2 \mapsto v
\end{align*} \]
```
This lecture's model of computation is often called the \textit{substitution model}.

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

\begin{itemize}
\item I/O, exceptions, mutation, concurrency, ...
\item we can build models of these things, but they aren’t as simple.
\item ... even modelling substitution was somewhat tricky
\end{itemize}

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

\begin{itemize}
\item we can use it to formally prove that, for instance:
  \begin{itemize}
  \item $\text{map } f \ (\text{map } g \ x) = \text{map } (\text{comp } f \ g) \ x$
  \item proof: by induction on the length of the list $x$, using the definitions of the substitution model
  \end{itemize}
\end{itemize}

It is \textit{not} useful for reasoning about execution time or space

\begin{itemize}
\item more complex models needed there
\end{itemize}
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 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:
  - \( \text{map } f \left( \text{map } g \, \text{xs} \right) = \text{map } (\text{comp } f \, g) \, \text{xs} \)
  - proof: by induction on the length of the list, using the definitions of the substitution model

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

- more complex models needed there

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
Church's mistake

substitute:

fun xs -> map (+) xs

for f in:

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

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

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

map is free here – it refers to a library function

the problem was that the value you substituted in had a \textit{free variable} (map) in it that was \textit{captured}. 

substitute:

\[
\text{fun } xs \rightarrow \text{map (+) } xs
\]

for \( f \) in:

\[
\text{fun } ys \rightarrow \\
\text{let } \text{map } xs = 0::xs \text{ in } \\
f (\text{map } ys)
\]

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

\[
\text{fun } ys \rightarrow \\
\text{let } z \ xs = 0::xs \text{ in } \\
(f \text{ fun } xs \rightarrow \text{map (+) } xs) \ (z \ ys)
\]

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)