S3: Programs as Data

CSci 2041:
Advanced Programming Principles

University of Minnesota,
Prof. Van Wyk,
Spring 2018

Recall

Recall a simple binary tree inductive datatype.

```
type int_bin_tree =
  | Leaf of int
  | Node of int_bin_tree * int_bin_tree
```

Inductive data types naturally represent these kind of structures.

Programs as data

Inductive datatypes also naturally represent more interesting kinds of data, namely

- arithmetic expressions,
- computer programs,
- proofs,
- logical systems, etc.

This representation often simplifies defining computations (as functions) over this data.

Simple arithmetic expressions

- Let's consider some simple arithmetic expressions, over integers with only addition and multiplication.
- Expressions for a very simple calculator, for example
  - 1
  - 2
  - 3+4
  - 4*2+3
  - 3*(8+2)
  - 4+0

Expressions as trees

- Instead of textually, we can represent expressions as trees. For example,

- Easy to evaluate these to integer values.

Operator precedence and associativity

- Operator precedence and associativity matter when translating from a linear textual representation to a tree-based, hierarchical representation.
- A tree representation encodes the precedence and associativity of the operators.
- So we don't need a constructor in our datatype for parenthesis.

Precedence, encoded

- Consider the expression trees: (drawn during lecture)

- We've just swapped the operator nodes.
- But the intention is clear, even without any representation of parenthesis in the tree.

Expressions as inductive datatypes

What do we need to design a datatype for our simple expressions?

- A name for the type
- The value constructors
  So, what are the different varieties of expressions?
- a type for the `of` part of our value constructors

See `arithmetic.ml` in public repo.

What about `eval`?

We need two new clauses for `Sub` and `Div`.

```
let rec eval e =
  match e with
  | Int v -> v
  | Add (l,r) -> eval l + eval r
  | Mul (l,r) -> eval l * eval r
  | Sub (l,r) -> eval l - eval r
  | Div (l,r) -> eval l / eval r
```

Let expressions

- Consider adding let expressions to our expression language.
- We may add a value constructor like the following:
  `| Let of string * expr * expr`
- We thus need a way to refer to these identifiers
  `| Id of string`
- We can then define expressions such as
  - `Let ("x", Int 5, Add (Int 4, Id "x"))`
- So, what happens to `eval`?

Code is developed in `expr_let.ml` in public repo.

evaluating let-expressions

- How can we evaluate
  `Let ("x", Int 5, Add (Int 4, Id "x"))`?
- How must `eval` change?
- We need to evaluate expressions given a certain context.
- This context is the "environment" which identifiers to values to be used in evaluation.
- Let's consider some example expressions and environments, in picture form.
- What is the type of the environment?
  What functions are needed for it?

Unbound identifiers

- How can we evaluate
  `Add (Int 4, Mul( Int 3, Id "x"))`?
- We can't, it has an unbound variable.

- Our extension to `eval` is in `expr_let.ml` in the code examples directory.
- It makes use of an additional argument to provide the appropriate environment when evaluating an expression.

Scope in let-expressions

Consider the following:

```
let nested = Let("x", Int 3,
                 Add(Mul (Int 2, Id "x"),
                     Let("x", Int 4,
                         Add(Int 5, Id "x"))))
```

- How do we distinguish between the two "x" identifiers?
- What is the scope of each declaration of x?
- Note how the simple list and process of searching from the beginning solves this problem in this simple language.

Adding relational and logical operators

- What do we need to do to add relational operators so that expressions such as `1 + 3 < 5` can be represented?
- What about logical operators?
- How is `eval` extended?
- How can we ensure that only type-correct expressions are evaluated?

  Expressions such as `3 + (4 < 5)` should be detected as ill-typed or not represent-able in our datatype.

  This last question is the interesting one.

One approach

Encode the well-formedness restriction in the datatype so that ill-formed expressions cannot be created.

- Recognize that logical expressions produce Boolean values from Boolean values.
- And that relational operations result in Boolean values, but operate on integer values.
- And that arithmetic operations consume and produce integer values.
- We can make this distinction in the OCaml types.
- Start over with two types: `int_expr` and `bool_expr`.


```
type int_expr =
  | Int of int
  | Add of int_expr * int_expr
  | Sub of int_expr * int_expr
  | Mul of int_expr * int_expr
  | Div of int_expr * int_expr
type bool_expr =
  | True
  | False
  | Lt of int_expr * int_expr
  | Eq of int_expr * int_expr
  | And of bool_expr * bool_expr
  | Or of bool_expr * bool_expr
  | Not of bool_expr
```


- How does our `eval` function need to change?
- We need two functions, one for `int_expr` and one for `bool_expr`.
- `eval_int_expr : int_expr -> int`
- `eval_bool_expr : bool_expr -> bool`

A problem with this approach

- So, we encoded the type (int or bool) of the expression in the type (`int_expr` or `bool_expr`) of the the expressions representation.
- What happens if we add let-expressions and variables?
- `let x = 3 + 4 in x + 5`
  or
  `let b = 3 < 5 in b && true`
- How can we represent these?
- `| Id of string`,
  but is this an `int_expr` or a `bool_expr`?
- `| Let1 of string * int_expr * int_expr` or
  `| Let2 of string * bool_expr * int_expr` or
  `| Let3 of string * bool_expr * bool_expr`
  or all? And what is its type?

A second, better approach

- Variables can be of any type and we can't easily determine this when the tree is constructed.
- Determining types is usually an analysis phase on the already constructed tree representation of the expression.
- Thus we fall back to one kind of expression, `expr`, but then construct trees that may have type errors in them, but we detect this in an analysis phase.

So, with just one type.

```
type expr =
  | Int of int
  | Add of expr * expr
  | Sub of expr * expr
  | Mul of expr * expr
  | Div of expr * expr
  | True
  | False
  | Lt of expr * expr
  | Eq of expr * expr
  | And of expr * expr
  | Not of expr
  | Let of string * expr * expr
  | Id of string
```

How can expression evaluation go wrong?

Before writing a new version of `eval`, how can things go wrong?

- undeclared names
- type errors
- division by zero

Can `eval` detect the problems **dynamically** and raise an exception if, for example, we try to evaluate `Add (Int 4, Bool true)` ?
(see `int_bool_exprs.ml`)

Can we detect any of these problems **statically**? That is, without trying to evaluate the expression.

Dynamic type checking

How must `eval` change to detect these problems?

Does the return type of `eval` change?

What might it be?

Name analysis, a static analysis

- The process of determining if there are any unbound variables in the expressions.
- What should the result of name analysis be? Perhaps a list of unbound names.
- What should name analysis produce for each of the following?
  - `Let ("x", Int 5, Add (Int 4, Id "x"))`?
  - `Add (Int 4, Mul( Int 3, Id "x"))`?

Type checking and type inference

- Our "language" so far is quite simple.
- Simple enough that we can infer types quite easily.
- Let's consider some examples.

Functions

We can easily represent functions in our inductive type by adding the following:

```
| App of expr * expr
| Lambda of string * expr
```

What does `let add = fun x -> fun y -> x + y` look like in our data type?

Type checking

- Can we always infer a simple type for functions?
  What about
  `let id = fun x -> x` ?
- How might we modify out language to support type checking?
- What is the "language of types" for our language?
- See `typed_exprs.ml` in the public class repository.

Values for functions

Let's focus on evaluation using functions and save type checking for later. Thus we'll revert to our simple language without types.
So far, values have been rather simple.

What is the value for

- `fun x -> fun y -> x + y`
- or for `add2` in the following

  ```
  let add2 =
    let two = 2 in
    fun x -> x + two
  ```

  Here the value of `add2` is a function, but with a small environment binding `two` to the value 2.
  This is called a closure.

Closures

A **closure** consists of

1. the name of the function parameter
2. the unevaluated body
3. an environment with bindings for all of the free variables in the body **except** for the function parameter

What might some examples of this be? Perhaps `add`?

See how curried functions simplify things here.

Recursive functions will pose some interesting challenges... stay tuned.

Recall: Values for functions

So far, values have been rather simple.

What is the value for

- `fun x -> fun y -> x + y`
- or for `add2` in the following

  ```
  let add2 =
    let two = 2 in
    fun x -> x + two
  ```

  Here the value of `add2` is a function, but with a small environment binding `two` to the value 2.
  This is called a closure.

Closures

A **closure** consists of

1. the name of the function parameter
2. the unevaluated body
3. an environment with bindings for all of the free variables in the body **except** for the function parameter

```
type value
  = ...
  | Closure of string * expr * environment
```

What might some examples of this be? Perhaps `inc` or `add`?

We'll see how curried functions simplify things here.

A comment

We'll write expressions in our `expr` type in double quotes instead writing them directly.

For example, ``` ``x + 1'' ``` is to be seen as

`Add (Id "x", Value (Int 1))`

But this ``` ``...'' ``` notation is more concise.

inc

- ``` let inc = ``fun x -> x + 1'' ```
- value of inc: ``` Closure ("x", ''x + 1'', []) ```
- Now, evaluate ``` ``inc 3'' ```
- evaluate ``` ``inc'' ``` $\longrightarrow$ ``` Closure ("x", ''x + 1'', [])```
  evaluate ``` ``3'' ``` $\longrightarrow$ `Int 3`
- now apply the function to the argument
  evaluate ``` ``x + 1'' ``` but what is its environment?
  it is the environment of ``` ``inc 3'' ``` but with `("x", Int 3)` added to it.
  so, evaluate ``` ``x + 1'' ``` in `[("x", Int 3)]`

let and lambdas

- In fact
  `‘‘let x = 3 in x + 1’’`
- is the same thing as
  `‘‘(fun x -> x + 1) 3’’`
- that is
  `let x = ... dexpr ... in ... body ...`
- is the same thing as
  `(fun x -> ... body ...) (... dexpr ...)`

add2

```
‘‘let add2 =
        let two = 2 in fun x -> x + two
  in add2 4’’
```

evaluating `add2` in `env` is first
`(("two",Int 2)::env) (’’fun x -> x + two’’)`
which becomes
`Closure ("x", ’’x + two’’, ("two",Int 2)::nil)`

So we looked at the free variables in the lambda expression and created an environment for them (and only them). We can just look them up in the environment.

Apply add2

Now apply it: `‘‘add2 4’’`

evaluate `‘‘add2’’`
⟶ `Closure ("x", ’’x + two’’, [("two",Int 2)])`

evaluate `‘‘4’’` ⟶ `Int 4`

this is
evaluate `‘‘x + two’’`
in environment `[("x", Int 4); ("two", Int 2)]`

add

```
‘‘let add = fun x -> fun y -> x + y
  in (add 1) 2’’
```

value of `add` is `Closure ("x", fun y -> x + y, [])`

Now, evaluate `‘‘add 1’’`
evaluate `add`
$\longrightarrow$ `Closure ("x", fun y -> x + y, [])`
evaluate `‘‘1’’` $\longrightarrow$ `Int 1`
now apply it, yielding
`Closure ("y", ‘‘x + y’’, [("x", Int 1)])`
apply this to `Int 2`
evaluate `‘‘x + y’’` in `[("y", Int 2); ("x", Int 1)])`

Recursive functions

How do we represent the closure for a recursive function?

```
‘‘let rec sumToN n =
    if n = 0 then 0 else n + sumToN (n-1)’’
```

Maybe

```
let c =
  Closure ("n",
           ‘‘if n = 0 then 0 else n + sumToN (n-1)’’,
           [ ("sumToN", ????) ] )}
```

But what about `sumToN`?
It should be a reference to `c` - the closure itself.

Circular structures

OK, so `c` is a value that somehow contains a reference to `c`.

How can we create such a thing?

We need some mechanism for doing this in OCaml.

We need OCaml references. So let's consider these in
S5.1 Imperative OCaml Programming.

Closures for recursive functions

```
‘‘let rec sumToN n =
    if n = 0 then 0 else n + sumToN (n-1)’’
```

We need a new kind of `value` like `Int` and `Closure`

```
| Ref of value ref
```

Then,

```
let recRef = ref (Int 999) in
let c =
  Closure ("n",
           ‘‘if n = 0 then 0 else n + sumToN (n-1)’’,
           [ ("sumToN", Rec recRef) ] ) in
let () = recRef := c in
  ...
```


- So we create a reference to a “dummy” `value`
- Then evaluate the lambda expression with an environment that contains the binding of the function name to this dummy value.
- The function name is a free variable in the function body.
- After we've evaluated the lambda expression to a value, we update the reference to point to this value.
- It creates a circular structure.

To summarize

We've seen a few different representations for expressions, growing in complexity.

Functions for evaluating or type checking expressions follow the inductive structure of the data.

What we've defined are “interpreters”. The execute the program directly. Compilers work by translating the program to some executable language (byte-codes or machine instructions).

Interpreters for mainstream languages are more sophisticated and include many optimizations, but this gives us a taste of how they work.