S5: Programs as Data

CSci 2041:

Advanced Programming Principles

University of Minnesota,
Prof. Van Wyk,
Spring 2022

Recall

Recall a simple binary tree inductive datatype.

```
type 'a tree =
  | Empty
  | Node of 'a * 'a tree * 'a 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.

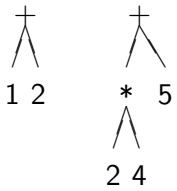
This representation often simplifies defining computations (as functions) over this data.

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 ← which operator is computed first?
  - 3*(8+2) ← parenthesis determine operator order
  - 4+0

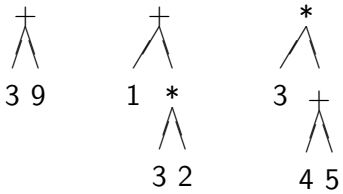
Expressions as trees

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



- Easy to evaluate these to integer values.

Exercise S5: #1: What do the following expression trees evaluate to?

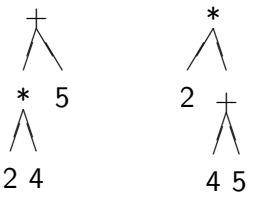


Operator precedence and associativity

- 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 with the same leaves: 2, *, 4, +, and 5



- We've just changed the structure.
- 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

Expressions as data

We've done this before in `StackMachine`.

```
type expr = Add of expr * expr
          | Sub of expr * expr
          | Mul of expr * expr
          | Div of expr * expr
          | Num of int

let rec eval (e: expr) : int = match e with
  | Add (e1, e2) -> eval e1 + eval e2
  | Sub (e1, e2) -> eval e1 - eval e2
  | Mul (e1, e2) -> eval e1 * eval e2
  | Div (e1, e2) -> eval e1 / eval e2
  | Num n -> n
```

See `expr.ml` in the `Sample Programs/Expressions` directory of the public repo.

Exercise S5: #2: Converting trees to strings

Write a function `string_of_expr` of type `expr -> string` that converts an expression to the strings we saw at the beginning of these slides.

Recall:

```
type expr = Add of expr * expr
          | Sub of expr * expr
          | Mul of expr * expr
          | Div of expr * expr
          | Num of int
```


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", Num 5, Add (Num 4, Id "x"))`
- So, what happens to `eval`?
- We need to evaluate expressions given a certain context.
- This context is the "environment" which identifiers to values to be used in evaluation.

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

Some example environments

How can we evaluate ths following?

```
Let ("x", Num 5, Add (Num 4, Id "x"))
```

- Evaluate the entire expression in an "empty" environment.

  e.g. `[]` - empty environment

- For the sub-expressions `Add (Num 4, Id "x")`

  use `[ ("x", 5) ]`

Some example environments

How can we evaluate this one?

```
Add (Num 4, Mul( Num 3, Id "x"))
```

1. If we don't have an environment, then we can't evaluate it.
2. If we have `[ ]` as the environment, then evaluation fails.

   The expression has an unbound variable.

The point:

- we need the environment, without we cannot evaluate an expression.
- the environment needs binding for all free variables for evaluation to succeed.

Some example environments

What about this?

```
Let( "x", Num 3,
     Add( Mul (Num 2, Id "x"),
          Let( "y", Add (Num 4, Id "x"),
                    Add (Id "x", Id "y"))))
```

As a string this might be

```
"let x = 3 in (2 * x) + (let y = 4 + x in x + y)"
```

- for the entire expression, again use `[]`
- for the outermost add expression (the body of the outermost let) and the addition of `4` and `x`, use

  `[ ("x", 3) ]`
- for the innermost add expressions (the body of the innermost let), use

  `[ ("y", 7); ("x", 3) ]`

  We keep both bindings. The closest (inner most) binding is at the front and the farthest away (outer most) is at the end of the list.

Scope in let-expressions

What if we re-name `y` to `x` in the previous example?

```
Let( "x", Num 3,
     Add( Mul (Num 2, Id "x"),
          Let( "x", Add (Num 4, Id "x"),
                    Add (Id "x", Id "x"))))
```

As a string this might be

```
"let x = 3 in (2 * x) + (let x = 4 + x in x + x)"
```

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

Scope in let-expressions

Recall the previous example:

```
Let( "x", Num 3,
     Add( Mul (Num 2, Id "x"),
          Let( "x", Add (Num 4, Id "x"),
                    Add (Id "x", Id "x"))))
```

As a string this might be `"let x = 3 in (2 * x) + (let x = 4 + x in x + x)"`

- for the entire expression, again use `[]`
- for the outermost `Add` expression (the body of the outermost let), use

  `[ ("x", 3) ]`
- for the second `Add` expression (the value bound to `y`), use

  `[ ("x", 3) ]`
- for the third `Add` expressions (the body of the second `Let`), use

  `[ ("x", 7); ("x", 3) ]`

  We keep both bindings, but use the first one in the list.

Implementing the environment

- What is the type of the environment?
- What functions are needed for it?
- Our extension to `eval` is in `let.ml` in the `expr` directory in the code examples directory.
- It makes use of an additional argument to provide the appropriate environment when evaluating an expression.

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 representable 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`.
- See `int_expr_bool_expr.ml`

- 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.

Values

We will also identify which expressions are values.

```
type expr
  = Val of value
  | Add of expr * expr
  | ...
  | And of expr * expr
  | ...
and  value
  = Int of int
  | Bool of bool
```

Exercise S5: #3:

Without rewriting the evaluation function `eval`, just write down how it might need to change again.

We'll develop a few ideas in `int_bool_expr.ml`.

Some constructors are given below:

```
type expr
  = Val of value
  | Add of expr * expr
  | Sub of expr * expr
  | Lt of expr * expr
  | And of expr * expr
  | Let of string * expr * expr
  | Id of string
and value
  = Int of int
  | Bool of bool
```


How can expression evaluation go wrong?

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

1. division by zero (this was possible in previous versions of `eval`)
2. undeclared names
3. type errors, e.g. ``` ``1 + true'' ```

When can these problems be detected?

- at evaluation time - that is **dynamically**
- before evaluation time - that is **statically**

Detecting undeclared names and type errors can easily be done statically.

But detecting division by zero is too restrictive and thus done dynamically.

Dynamic checking: for undeclared names and type errors

How must `eval` change to detect these problems?

- Values must have type information at evaluation time.
- This type information is checked before an operation is carried out.

The type of `eval` could change to any of the following:

- `expr -> value`: and then raise an exception for any problems detected during evaluation. We explored this in `int_bool_expr.ml`.
- `expr -> value option`: since we may not get a `value` for all expressions
  But this tells us little about failed evaluations.
- `expr -> value result`: where
  `type 'a result = Result of 'a | Error of string`

  This would return a value or an error message.

Static checking: checking for unbound names

- The "compile-time" process of determining if there are any unbound variables in 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"))`?

Exercise S5: #4:

If we are to write a "unbound name analysis" function,

1. What is its type?
2. Sketch its definition for a few interesting cases of `Expressions`.

Free variables

A function that checks for "free-variables" is a solution to our "unbound name analysis" problem.

For example: `freevars : expr -> string list`

The free variables in an expression are the variables that are not bound, **in that expression**.

- `freevars (Add (Val (Int 1), Val (Int 2))) = [ ]`
- `freevars (Add (Val (Int 1), Id "x")) = [ "x" ]`
- `Let ("x", Val (Int 6), Add (Val (Int 1), Id "x")) = [ ]`
- In `Let ("x", Val (Int 6), Add (Val (Int 1), Id "x"))`

  we still say that

  `freevars (Add (Val (Int 1), Id "x")) = [ "x" ]`

  The `Let` construct would remove the name it binds from the free variables of the body of the let-expression.

Type Checking and Type Inference

Type checking "checks" that the types given to identifiers in the code match how they are used.

- To do this, we need to change the `Let` constructor to include the type of the name:

  ```
  | Let of string * typ * expr * expr
  ```

  where `typ` is specifies types.
- see `let_type_check.ml`

Type inference "infers" the types of identifies based on how they are used. Thus we do not write down types in the code.

- Thus we can keep `Let` as we had it before:

  ```
  | Let of string * expr * expr
  ```
- see `let_type_infer.ml`

But first consider the exercise on the next page.

Exercise S5: #5:

1. For our small language with let-expressions,

   what is the type of a function `type_check` that does type checking?

2. For our small language with let-expressions,

   what is the type of a function `type_infer` that does type inference?

Adding functions to our language

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

```
| App of expr * expr
| Fun of string * expr
```

or if we want to do type checking:

```
| App of expr * expr
| Fun of string * typ * expr
```

(`App` = application, `Fun` = lambda-expressions, function literal)

What does

```
let add = fun x -> (fun y -> x + y)
```

look like in our first data type above?

```
Let ("add", Fun ("x",
                 Fun ("y", Add (Id "x", Id "y"))
                )
    )
```

Type checking

Can we always infer a simple type for functions?

- What about `let inc = fun x -> x + 1` ?

  Maybe `FunT (IntT, IntT)`.

- What about `let add = fun x -> (fun y -> x + y)` ?

  Maybe `FunT (IntT, Fun (IntT, IntT))`

- What about `let id = fun x -> x` ?

  Something like `'a` in OCaml.

We'll skip type checking for functions and spend our time on their evaluation.

Evaluation of functions

How do we represent functional values?

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`.

The challenge is that the definition of `two` no longer exists when we use the value of `add2`.

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`?

We'll see how curried functions simplify things here.

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

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`?

A comment

We'll write expressions in our `Expressions` 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 using ``` `` ``` and ``` '' ``` is more concise.

An increment function

- `let inc = ``fun x -> x + 1''`
- The value of `inc` is `Closure ("x", ``x + 1'', [])`

  That is, `eval 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.

    So, evaluate ``` ``x + 1'' ``` but what is its environment?

    It is
    - the environment from the closure, that is `[]`
    - with the binding of the function argument

    That is `[("x", Int 3)]`.
  - So, evaluate ``` ``x + 1'' ``` in `[("x", Int 3)]`

Let- and lambda-expressions

- 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 ...)

Functions with free variables: add2

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

In the last line, evaluating `eval add2 ...` leads to

`eval ``let two = 2 in fun x -> x + two'' ...`

which leads to

`eval ``fun x -> x + two'' ([("two", Int 2)] @ ...)`.

This becomes

`Closure ("x", ``x + two'', [("two",Int 2)])`

The free variables in the lambda expression **body** are used to create an environment (of them and only them). We can just look them up in the environment.

Application of add2

Now apply it: ``add2 4''

First evaluate ``add2''
$\longrightarrow$ `Closure ("x", ''x + two'', [("two",Int 2)])`

Then evaluate ``4''
$\longrightarrow$ `Int 4`

Then evaluate ``x + two''

in environment `[("x", Int 4); ("two", Int 2)]`

This gives us `Int 6`.

Curried functions: add

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

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

Now, evaluate ``add 1''.

First evaluate `add` $\longrightarrow$ `Closure ("x", ``fun y -> x + y'', [])`

Then evaluate ``1'' $\longrightarrow$ `Int 1`

Now applying `Closure ("x", ``fun y -> x + y'', [])` to `Int 1` yields
`Closure ("y", ``x + y'', [("x", Int 1)])`.

Apply this to `Int 2` yields
``x + y'' in `[("y", Int 2); ("x", Int 1)])`.

Evaluation of the above yields 3, as expected.

Recursive functions

How do we represent the closure for a recursive function?

```
``let rec sumToN = fun n ->
    if n = 0 then 0 else n + sumToN (n-1)
  in sumToN 10''
```

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
S6.1 Imperative OCaml Programming.

Closures for recursive functions

```
``let rec sumToN = fun n ->
    if n = 0 then 0 else n + sumToN (n-1)
  in sumToN 10''
```

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", Ref 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.