CSci 2041
Advanced Programming Principles
L3.2: Recursive computations over recursive data
Programs as data, expressions

Prof. Eric Van Wyk

Fall 2014

Recall

Recall a binary tree disjoint union.

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

Algebraic data types naturally represent these recursive structures.

Programs as data

Disjoint unions also naturally represent more interesting kinds of data, namely

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

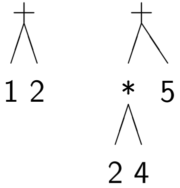
This representation often simplifies defining computations (as functions) over this data.

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,

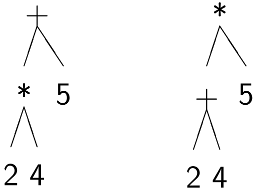


- Easy to evaluate these to integer values.

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 disjoint union for parenthesis.

- Consider the expression trees:



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

Expressions as disjoint unions

What do we need to design an disjoint union 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

What about `eval`?

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

```
let rec eval e =
  match e with
  | Add (l,r) -> eval l + eval r
  | Mul (l,r) -> eval l * eval r
  | Const v -> v
  | 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
  `| Var of string`
  (We might consider calling these "identifiers" instead of "variables" since their values never change.)
- We can then define expressions such as
  - `Let ("x", Const 5, Add (Const 4, Var "x"))`
- What happens to `eval`?

- How can we evaluate `Let ("x", Const 5, Add (Const 4, Var "x"))`?
- How must `eval` change?
- We need to evaluate expressions given a certain context.
- This context is the "environment" which maps variable names 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?

- How can we evaluate `Add (Const 4, Mul( Const 3, Var "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 approriate environment when evaluating an expression.
- This is the same pattern that we saw in the `show_btree` function. (In `btree.ml` in the code examples.)

Scope in let-expressions

Consider the following:

```
let nested = Let("x", Const 3,
                 Add(Mul (Const 2, Var "x"),
                     Let("x", Const 4,
                        Add(Const 5, Var "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 representable in our disjoint unions.

  This last question is the interesting one.

One approach

Encode the well-formedness restriction in the disjoint union 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 =
  | Add of int_expr * int_expr
  | Mul of int_expr * int_expr
  | Const of int
  | Sub of int_expr * int_expr
  | Div of int_expr * int_expr
 and bool_expr =
  | 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
```

Note the use of “and” for mutually recursive types.

- How does our `eval` function need to change?
- We need two functions, one for `int_expr` and one for `bool_expr`.
- `int_expr` returns an `int`
- `bool_expr` returns an `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?
- `| Var of string`,
  but is this an `int_expr` or a `bool_expr`?
- `| Let of string * int_expr * int_expr` or
  `| Let of string * bool_expr * int_expr` or
  `| Let of string * bool_expr * bool_expr`
  or all? And what is its type?

A “second” 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 =
  | Add of expr * expr
  | Mul of expr * expr
  | Const of int
  | Sub of expr * expr
  | Div of expr * expr
  | Lt of expr * expr
  | Eq of expr * expr
  | And of expr * expr
  | Or of expr * expr
  | Not of expr
  | Let of string * expr * expr
  | Var 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 we detect any of these problems statically? That is, without trying to evaluate the expression.

Name 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 undeclared names.
- What should name analysis produce for each of the following?
  - `Let ("x", Const 5, Add (Const 4, Var "x"))`?
  - `Add (Const 4, Mul( Const 3, Var "x"))`?

Type checking

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