Language Design

COMP 320

Goals

What is the meaning of a program?

What are the legal expressions?

What is the concept of a variable?

What is the scope of a variable?

When is an expression well typed?

Does every expression have a unique type?

A Small Language

We start with a small subset of ML

The set of expressions is defined inductively as:

1. A number is an expression
2. The booleans **true** and **false** are expressions
3. If e1 and e2 are expressions, then e1 op e2 is an expression where op ∈ {+, -, *, =, <, orelse}
4. If e0, e1 and e2 are expressions then **if e0 then e1 else e2** is an expression

BNF

Backus-Naur-Form (BNF) provides a compact way of defining expressions:

op ::= + | - | * | = | < | orelse

e ::= n | true | false | e op e | if e then e else e | (e)

This grammar defines the set of well-formed expressions

An expression is any string that can be derived from this grammar

(We will not provide a formal definition of derivations here)

Well Formed Expressions

Examples of well-formed expressions

```
 3 + (2 + 4)

true + (2 + 4)

if (2 = 0) then 5 + 3 else 2

if true then (if false then 5 else 1 + 3) else 2 = 5

 (if 0 then 55 else 77 - 23) = 0
```

These expressions are syntactically well formed but the second and the last are not well-typed

Syntactically Invalid Expressions

Examples of ill-formed expressions

```
if true then 2 else
```

```
-4
```

```
+23
```

These expressions may “make sense” but they are not well formed based on our grammar

The grammar requires that every operator has two arguments

Operational Semantics

We begin to describe in formal terms what it means to evaluate an expression in our tiny language

Notation: $e \downarrow v$ means that the expression, e, evaluates to the final value v

The final values in our language are numbers, true or false

We can evaluate an expression by recursively evaluating all of its subexpression and then determining the value of the original expression.

For example: To evaluate e1 * e2

1. Evaluate e1 to get some value v1
2. Evaluate e2 to get some value v2
3. Compute the value v1 * v2 using the multiplication operation

Formal Definition

We can give a formal definition of this process using inference rules.

In general, an inference rule has the components:

1. A set of premises
2. A conclusion
3. (Possibly) a name for the rule

Examples

The rules for numbers and Booleans have no premises and have the conclusions n↓n, true↓true, false↓false

The operational semantics of the expression we looked at, e1 * e2, has premises e1↓v1 and e2↓v2 with conclusion e1 * e2↓v1*v2

We can give similar evaluation rules for other expressions such as if's

Using these rules we can evaluate an expression such as

```
if ((4-1)<6) then 3+2 else 4 ↓ 5
```

What about

```
if 6 then 3+2 else 4
```

Properties

The successful evaluation of an expression will yield a value.

This property is called **value soundness**

The value is unique.

This property is called **determinancy**

It is possible to formally prove these properties by structural induction

Concrete and Abstract Syntax

The BNF grammar for expressions defines a concrete syntax for syntactically valid expressions

To evaluate and process these expressions, they are first converted into abstract syntax trees

This processes done by lexical analysis and parsing

We will assume that this processing has been done and we are dealing with the abstract syntactic representation of expressions

These trees can be represented by SML datatypes

Abstract Syntax

```
(* Primitive operations *)
datatype primop = Equals | LessThan | Plus
                   | Minus | Times | OrElse
(* Expression *)
datatype exp =
      Int of int                (* n *)
    | Bool of bool              (* true or false *)
    | If of exp * exp * exp     (* if e then e1 else e2 *)
    | Primop of primop * exp * exp  (* e1 <op> e2 *)
```

Evaluation of Primitive Ops

```
fun evalOp (Equals, [Int i, Int i']) = SOME (Bool (i = i'))
  | evalOp (LessThan, [Int i, Int i']) = SOME (Bool (i<i'))
  | evalOp (Plus, [Int i, Int i']) = SOME (Int (i + i'))
  | evalOp (Minus, [Int i, Int i']) = SOME (Int (i - i'))
  | evalOp (Times, [Int i, Int i']) = SOME (Int (i * i'))
  | evalOp (OrElse, [Bool a, Bool a1]) =
     SOME (if a then (Bool true) else eval a1)
  | evalOp _ = NONE
```

An Evaluator

```
eval (Int(n)) = Int(n)
  | eval (Bool(b)) = Bool(b)
  | eval (Primop(po, arg1, arg2)) =
   (case evalOp(po, eval(arg1), eval(arg2)) of
      NONE => raise Stuck "Unexpected arguments to primitive operations"
    | SOME(v) => v)
  | eval (If(e,e1,e2)) =
   (case eval(e) of
      Bool(true) => eval(e1)
    | Bool(false) => eval(e2)
    | _ => raise Stuck "Expected boolean value")
  | eval _ = raise notImplemented
```

Well-typed expressions

Even within our very tiny language there are many expressions which cannot be evaluated

Our interpreter would get stuck and raise an exception

Our operational semantics would break down as well

Examples;

```
    7 + true

    if 7 then 45 else 72
```

Type Checking

Type checking approximates the runtime behaviour

It allows us to detect errors statically, early in the development of the program

We look at our very simple language first before extending it to include variables, let expressions and functions

Type Checking

Recall our grammar:

```
op ::= + | - | * | = | < | orelse
```

```
 e ::= n | true | false | e op e | if e then e else e
| (e)
```

There are only two basic types, int and bool

We denote these as

```
      Types T : int | bool
```

The type int is made up of numbers and the type bool of the values true and false

Types

We use the following notation:

```
e:T means expression e has type T
```

To formally define the type of an expression we use structural induction on the structure of the expression

We use the notation of premises and conclusions that was introduced above in formalizing the operational semantics

The type rules for the primitive values have no premises but only conclusions and we can name the rules for convenience:

```
n : int (T-NUM)
true : bool (T-TRUE)
false : bool (T-FALSE)
```

Type Inference Rules

When is e1+e2 well typed?

```
Premises: e1 : int, e2: int
Conclusion e1+e2 : int
Name: T-PLUS
```

We have similar rules for each of the primitive operations

For if expressions we have:

```
Premises: e:bool, e1:T, e2:T
Conclusion: if e then e1 else e2:T
Name: T-IF
```

Adding Variables and Let

We extend our language by adding two productions to the BNF grammar:

```
    e ::= … | x | let x=e in e end
```

Syntactically correct expressions:

```
  let z = if true then 2 else 43 in z + 123 end
  let z = if 7 then 2 else 43 in z + false end
  let x = x + 3 in y + 123 end
  let x = x + 3 in x + 123 end
```

Syntax Errors

```
let z = if true then 2 else 43 in - 123 end
```

The unary operator “-” is not part of the grammar.

```
let x = 3 in x
```

The keyword ”end” is missing

```
let x = 3 x + 2end
```

The keyword “in” is missing

Bindings

In defining the operational semantics we have to deal with variable bindings

When is a variable bound to a value?

When is it free?

Free Variables

For a given expression, we can define the set of free variables that occur in the expression

```
FV(x) = {x}

FV(e1 op e2) = FV(e1) ∪ FV(e2)

FV(if e then e1 else e2) =

                  FV(e) ∪ FV(e1) ∪ FV(e2)

FV(let x = e1 in e2 end) =

                  FV(e1) ∪ (FV(e2)/{x})
```

(where / represents the set difference operation)

Bindings

Bindings are introduced by let expressions

The expression `let x = e1 in e2 end` introduces a binding which binds all of the free occurrences of x in e2

An expression e which has no free variables is called **closed**

Example

`let x=5 in (let y = x + 3 in x + y end) end`

Subexpressions:

`let x=5 in (let y = x + 3 in` **`x + y`** `end) end`

x and y are free

`let x=5 in (let y =` **`x + 3`** `in x + y end) end`

x is free

`let x=5 in (`**`let y = x + 3 in x + y end`**`) end`

x is free, y is bound

`let x=5 in (let y = x + 3 in x + y end) end`

x and y are bound

Example 2

```
let x=5 in (let x = x + 3 in x + x end) end
```

We number occurrences of x:

```
let x1=5 in (let x2 = x3 + 3 in x4 + x5 end) end
```

Then, in `x4 + x5`, both x's are free

In `(let x2 = x3 + 3 in x4 + x5 end)`, x4 and x5 are bound by the binding of x2. The occurrence x3 is free

In the overall expression, x3 is bound by the binding of x1

Renaming

Names do not matter for bound variables

Their names can be changed:

```
let x=5 in (let x = x + 3 in x + x end) end
```

Is equivalent to

```
let x=5 in (let y = x + 3 in y + y end) end
```

Substitution

In evaluating a let-expression, we substitute the value bound to the variable for the variable in the body of the expression

To describe the semantics of evaluating let-expressions we need to formalize the notion of substitution. We use the notation `[e'/x]e` to represent the result of replacing all free occurrences of x in e with e’

```
 [e'/x](x)  = e'

 [e'/x](e1 op e2)  = [e'/x]e1 op [e'/x]e2

 [e'/x](if e then e1 else e2)  =

          if [e'/x]e then [e'/x]e1 else [e'/x]e2
```

Substitution in let-expressions

```
[e'/x](let y = e1 in e2) = ???
```

How about

```
let y = [e'/x]e1 in [e'/x]e2)
```

Example:

```
[5/x](let y = x + 3 in y + x end)
= let y = [5/x](x + 3) in [5/x](y + x) end
= let y = 5 + 3 in y + 5 end
```

But …

Problem

```
[y + 1/x](let y = x + 3 in y + x end)
= let y = [y + 1/x](x + 3) in [y + 1/x](y + x) end
= let y = (y + 1) + 3 in y + (y + 1) end
```

But this seems wrong, since y occurred free in the term y+1, but becomes bound

This is called variable capture

To avoid this, we use the property that names of bound variables don’t matter

Solution

Since names of bound variables don’t matter, we can rename the bound variable y to z:

```
 [y + 1/x](let y = x + 3 in y + x end)

= [y + 1/x](let z = x + 3 in z + x end)

= let z = [y + 1/x](x + 3) in [y + 1/x](z + x) end

= let z = (y + 1) + 3 in z + (y + 1) end
```

Substitution Rule for Let

We can define a substitution rule for let with the restriction that the variable of the let-expression does not occur free in the body (so that it won't capture anything)

We can achieve this by renaming bound variables

```
 [e'/x](let y = e1 in e2) =
                 let y = [e'/x]e1 in [e'/x]e2),
      provided that x≠y and y is not in FV(e')
```

This problem of variable capture will also occur in function application

Inference Rule For Let

For the let-expression we have:

Premises:

1. `e1 evaluates to v1`
2. `[v1/x]e2 evaluates to v`

Conclusion

- `let x = e1 in e2 end evaluates to v`

Inference Rule example

```
let x=5 in (let y = x + 3 in x + y end) end
```

Evaluates to 13

We can write a formal proof using our inference rules

Adding Functions

We extend our language by adding productions to the BNF grammar:

```
    e ::= … | fn x => e | e1 e2 | rec f => e
```

While these productions are just syntactic, they are introduced to allow us to add certain semantic features to the language.

The first of these productions supports the definition of anonymous functions

The second production supports function application

The third allows us to define recursive functions. Because anonymous functions have no names, they do not support recursion. We have added this by means of the third function

Free Variables

In order to give inference rules for evaluating these new expressions, we will have to extend the definition of free variables

```
FV(e1 e2) = FV(e1) ∪ FV(e2)

FV(fn x => e) = FV(e)/{x}

FV(rec f => e) = FV(e)/{f}
```

(where / represents the set difference operation)

The anonymous function definition binds the function parameter

The recursive function definition binds the name of the function

Substitution Rules

The substitution rules must be formulated to avoid the problem of variable capture

```
[e'/x](e1 e2) = [e'/x]e1 [e'/x]e2

[e'/x](fn y => e) = fn y => [e'/x]e

      provided that x≠y and y is not in FV(e’)

[e'/x](rec f => e) = rec f => [e'/x]e

      provided that x≠f and f is not in FV(e’)
```

Evaluation Rules

Function definition:

Functions are evaluate to themselves so there is no premise and the conclusion is that the value of

```
fn x => e
```

Is just

```
fn x => e
```

Evaluation Rules

Function application:

The rule for `e1 e2`:

Premises:

```
1. e1 ↓ fn x => e
2. e2 ↓ v2
3. [v2/x]e ↓ v
```

Functions are unevaluated so the conclusion is:

```
fn x => e
```

Is just

```
fn x => e
```

Recursive Functions

The recursive rule mimics the SML definition. In order for a function to "call itself", it must have a name by which it can refer to itself. This is achieved by using a recursive value binding, which are ordinary value bindings qualified by the keyword rec.

SML provides recursive bindings of the form:

```
val rec var:typ = val
```

The right-hand side must be a function expression. Example:

```
 val rec factorial : int->int =
          fn 0 => 1 | n:int => n * factorial (n-1)
```

We are more familiar with the equivalent fun form of

```
fun factorial 0 = 1
  | factorial (n:int) = n * factorial (n-1)
```

Evaluation Rule for Rec

The rule for `rec f => e:`

Premises:

1. `[rec f => e/f] e ↓ v`

Conclusion:

```
[rec f => e/f] e ↓ v
```

Note: This is a high level description that supports reasoning about programs. It implies that the code for the function has to be there when the function is applied. This is not how recursion is implemented in practice

Example

Consider the following program:

```
sum = rec f=> fn x => if x = 0 then 0 else x+f(x - 1)
```

Write sum' for the first unfolding of the recursion:

```
sum' = fn x => if x = 0 then 0 else x + sum(x - 1)
```

How does evaluation proceed?