Concepts of Programming Languages

Prof. Dr. Guido Salvaneschi

Software Technology Group
TU Darmstadt | FB Informatik

FUNCTIONS

A Note on Abstraction

- A powerful concept in Computer Science
  - “Hide details”
  - “Make smth. generic w.r.t. smth. else”
  - “Parametrise over smth”

- Let expressions
  - Parametrise an expression based on another expression
- Functions
  - Parametrise over data (only?)
- Generics
  - Parametrise over types

A Taxonomy of Functions

- **First-class functions**
  - Functions are values/objects with all the rights of other values
    - Can be constructed at runtime
    - Can be passed as arguments to other functions, returned by other functions, stored in data structures etc
  - No first-class functions => functions can only be defined in designated portions of the program, where they are given names for use in the rest of the program
- **Higher-order functions**
  - Functions that return and/or take other functions as parameters
  - Parameterize computations over other computations
- **First-order Functions**
  - Functions that neither return nor take other functions as parameters
  - Parameterise computations over data

A Taxonomy of Functions

Some languages achieve some king of higher-orderless without first class functions

- Functions pointers (C, C++)
- Objects with a “call” method
  - “A function is an object with a single method”
- Eval

F1WAE

- A language with **first-order** functions
  - Function applications are expressions
- No first-class/higher-order functions:
  - Function definitions are **not** expressions

$\Rightarrow$ separate definitions from expressions

- predefined functions given to interpreter as an argument

Concrete & Abstract Syntax for F1WAE

```
<F1LAE> ::= <num>
          | {+ <F1LAE> <F1LAE>}
          | {- <F1LAE> <F1LAE>}
          | {let {<id> <F1LAE>} <F1LAE>}
          | <id>
          | ???
```

how does the concrete syntax change?

```scala
sealed abstract class F1LAE
case class Num(n: Int) extends F1LAE
case class Add(lhs: F1LAE, rhs: F1LAE) extends F1LAE
case class Sub(lhs: F1LAE, rhs: F1LAE) extends F1LAE
case class Let(name: Symbol, namedExpr: F1LAE, body: F1LAE) extends F1LAE
case class Id(name: Symbol) extends F1LAE
???
```

how does the abstract syntax change?

Concrete & Abstract Syntax for F1LAE

```
<F1LAE> ::= <num>
          | {+ <F1LAE> <F1LAE>}
          | {- <F1LAE> <F1LAE>}
          | {let {<id> <F1LAE>} <F1LAE>}
          | <id>
          | {<id> <F1LAE>}
```

concrete syntax for function application

What does this tell us about valid F1LAE programs?

```scala
sealed abstract class F1LAE
case class Num(n: Int) extends F1LAE
case class Add(lhs: F1LAE, rhs: F1LAE) extends F1LAE
case class Sub(lhs: F1LAE, rhs: F1LAE) extends F1LAE
case class Let(name: Symbol, namedExpr: F1LAE, body: F1LAE) extends F1LAE
case class Id(name: Symbol) extends F1LAE
case class App(funName: Symbol, arg: F1LAE) extends F1LAE
```

abstract syntax for function application

F1WAE

- Function applications can be nested.
- Function applications can be bound to names in let expressions.
- Function Id can be bound to names in let expressions.
- We cannot define new functions in F1WAE.
  Function definitions are not expressions.
  There is no expression of the kind {fun ...}

Function Definitions in F1LAE

- Cannot define functions in F1LAE
  - More strict than necessary for first-orderness
- Predefined functions are passed to the interpreter
- How to represent functions?

```
case class FunDef(argName: Symbol, body: F1LAE)
type FunDefs = Map[Symbol, FunDef]
```

- Example

```
FunDef('n, Add('n, 1))
```

- Class **`FunDef`** does not extend class **`F1LAE`** => no syntax for function definitions

Example Interpreter Calls

`scala.Symbol`: tick (') followed by identifier

```
interp(
    App('f, 10),
    Map('f -> FunDef('n, App('n, 'n)))
)
```

```
interp(
    App('f, 10),
    Map(
      'f -> FunDef('x, App('g, Add('x, 3))),
      'g -> FunDef('y, Sub('y, 1)))
```

Discussion

What is needed to interpret F1LAE expressions?

<Interpreter>

F1WAE Interpreter

```
def interp(expr: F1LAE, funDefs: Map[Symbol, FunDef]): Int = expr match {
  case Num(n) =
    n
  case Add(lhs, rhs) =>
    interp(lhs, funDefs) + interp(rhs, funDefs)
  case Sub(lhs, rhs) =>
    interp(lhs, funDefs) - interp(rhs, funDefs)

  case Let(id, expr, body) =>
    val body = subst(body, id, Num(interp(expr, funDefs)))
    interp(body, funDefs)

  case Id(name) => sys.error("found unbound id " + name)

  case App(fun, arg) => funDefs(fun) match {
    case FunDef(param, body) =>
      interp(subst(body, param, Num(interp(arg, funDefs))), funDefs)
  }
}
```

Demo

- The F1LAEImmediateSubstInterp interpreter

Discussion on Scoping

What is the result of interpreting **`{f 10}`** where **`{f n} = {n n}`**?

```
interp(
    App('f, 10),
    Map('f -> FunDef('n, App('n, 'n)))
)
```

- Should the interpreter try to substitute the **`n`** in function position of App with **`10`**? What would happen if this is the case?
- Or should function names and function arguments live in separate "spaces" ==> an identifier in function position within an App is not replaced.

Let's see what our interpreter does…

Discussion on Scoping: Namespaces

- Should the interpreter try to substitute the n in the function position of the application with the number 10, then complain that no such function can be found (or even that the lookup fails because the names of functions must be identifiers, not numbers)?
- (f 10) -> (interp (subst (app ‘n (id ‘n)) n 10)) -> (interp (app 10 10)) -> either “10: no such function definition” or “10 is invalid function name”

- **Single namespace** (e.g. Scheme). The name of a function can be bound to a value in a local scope, thereby rendering the function inaccessible through that name.

- **Multiple namespaces** (e.g., Common Lisp). The interpreter decide that function names and function arguments live in two separate “spaces”, and context determines in which space to look up a name.
  - An identifier in a function position within an app-expr is not replaced.
  - F1WAE employs namespaces.
  - Running F1WAE, the result is “n: : no such function definition”. Identifiers (incl. function arguments) and function names are “looked-up” differently (identifiers are not looked-up at all but immediately substituted as soon as a corresponding binding instance is found for them).

Discussion on Scoping

What is the result of `{f 10}` where `{f x} = {g {+ x 3}}`
`{g y} = {- y 1}`

```
val funDefs = Map(
    'f -> FunDef('x, App('g, Add('x, 3))),
    'g -> FunDef('y, Sub('y, 1))
)

interp(App('f, 10), funDefs)
```

- Should **`f`** be able to invoke **`g`** or should the invocation fail because **`g`** is defined after **`f`**?
- What if there are multiple bindings for the same name?
- If a function can invoke every defined function, it can also invoke itself. Do we have recursion in F1LAE?

Outline

- Implementing first-order functions
- **Environments (static vs. dynamic scoping)**
- About first-class functions
- Functional decomposition and recursion patterns
- Implementing first-class functions

`let` and Substitution

- In `{let {x e} t}` we immediately replace free identifiers `x` in expression `t` with the value expression `e` evaluates to
- `id` expressions left in `t` after substitution denote free identifiers
- If the interpreter encounters an `id` expression → error

Quiz
Do you see any problems with this strategy?

`let` and Environments

- The interpreter receives a store called **environment,** which maps identifiers to values
- `{let {x e} t}` simply stores in the **environment** a mapping from `x` to the value `e` evaluates to
- When the interpreter encounters an `id` expression, it looks up the corresponding value in the environment
- Free variables not in environment → error

- We represent environments as values of the type **`Env`**:

```
type Env = Map[Symbol, Int]
```

Here is F1LAE. How Does it Change?

```scala
def interp(expr: F1LAE, funDefs: FunDefs): Int = expr match {
    case Num(n) =>
      n
    case Add(lhs, rhs) =>
      interp(lhs, funDefs) + interp(rhs, funDefs)
    case Sub(lhs, rhs) =>
      interp(lhs, funDefs) - interp(rhs, funDefs)

    case Let(id, expr, body) =>
      val body = subst(body, id, Num(interp(expr, funDefs)))
      interp(body, funDefs)

    case Id(name) =>
      sys.error("found unbound id " + name)

    case App(fun, arg) => funDefs(fun) match {
      case FunDef(param, body) =>
        interp(subst(body, param, Num(interp(arg, funDefs))), funDefs)
    }
  }
```

Is this our F1LAE with Environments?

```
def interp(expr: F1LAE, funDefs: FunDefs, env: Env): Int = expr match {
    case Num(n) =>
      n
    case Add(lhs, rhs) =>
      interp(lhs, funDefs, env) + interp(rhs, funDefs, env)
    case Sub(lhs, rhs) =>
      interp(lhs, funDefs, env) - interp(rhs, funDefs, env)

    case Let(id, expr, body) =>
      val newEnv = env + (id -> interp(expr, funDefs, env))
      interp(body, funDefs, newEnv)

    case Id(name) =>
      env(name)

    case App(fun, arg) => funDefs(fun) match {
      case FunDef(param, body) =>
        val funEnv = env + (param -> interp(arg, funDefs, env))
        interp(body, funDefs, funEnv)
    }
}
```

What's the result of evaluating `{let {n 5} {f 10}}` where `{f x} = {n}`?

What is the answer when using F1WAE with substitution?

Interpreter

- This was the F1LAEDynamicInterp

Static Versus Dynamic Scoping

Definition Scope (of a name binding):
The scope of a name binding is the part of the program where the binding is in effect.

Definition Static/Lexical Scoping:
The scope of a name binding is determined syntactically (at compile-time).

Definition Dynamic Scoping:
The scope of a name binding is determined by the execution context (at runtime).

F1LAE with Environments

```
def interp(expr: F1LAE, funDefs: FunDefs, env: Env): Int = expr match {
    case Num(n) =>
      n
    case Add(lhs, rhs) =>
      interp(lhs, funDefs, env) + interp(rhs, funDefs, env)
    case Sub(lhs, rhs) =>
      interp(lhs, funDefs, env) - interp(rhs, funDefs, env)

    case Let(id, expr, body) =>
      val newEnv = env + (id -> interp(expr, funDefs, env))
      interp(body, funDefs, newEnv)

    case Id(name) =>
      env(name)

    case App(fun, arg) => funDefs(fun) match {
      case FunDef(param, body) =>
        val funEnv = env + Map(param -> interp(arg, funDefs, env))
        interp(body, funDefs, funEnv)
    }
  }
```

Static Scoping!

Interpreter

- This was the F1LAEStaticInterp