Principles of Programming Languages
Lecture 6: Functions. Default data structures.

Andrei Arusoaie[1]

[1]Department of Computer Science

November 7, 2017

Outline

Outline

Functions

- Functions: a way to organize our code into units

Functions

- Functions: a way to organize our code into units
- Input parameters, return value

Functions

- Functions: a way to organize our code into units
- Input parameters, return value
- Call-by-value: pass a copy of the original object

Functions

- Functions: a way to organize our code into units
- Input parameters, return value
- Call-by-value: pass a copy of the original object
- Call-by-reference: pass a reference to the original object

Functions in K: desired usage

Functions in K: desired usage

- Function *declaration*:

Functions in K: desired usage

- Function *declaration*:

```
function inc (int x) {
      x = x + 1;
      return x;
}
```

Functions in K: desired usage

- Function *declaration*:

```
function inc (int x) {
        x = x + 1;
        return x;
}
```

- Function *call*:

Functions in K: desired usage

- Function *declaration*:

```
function inc (int x) {
      x = x + 1;
      return x;
}
```

- Function *call*:

```
inc(2);
```

Functions in K: desired usage

- Function *declaration*:

```
function inc (int x) {
      x = x + 1;
      return x;
}
```

- Function *call*:

```
inc(2);
inc(x);
```

Functions in K: desired usage

- Function *declaration*:

```
function inc (int x) {
      x = x + 1;
      return x;
}
```

- Function *call*:

```
inc(2);
inc(x);
inc(x + y);
```

Programs: list of declarations

Programs: list of declarations

- Programs:

Programs: list of declarations

- Programs:

```
int a;
```

Programs: list of declarations

- Programs:

```
int a;
int b;
```

Programs: list of declarations

- Programs:

```
int a;
int b;
function inc (int x) {
      x = x + 1;
      return x;
};
```

Programs: list of declarations

- Programs:

```
int a;
int b;
function inc (int x) {
      x = x + 1;
      return x;
};
function main () {
      b = inc(a);
      return;
}
```

Programs: syntax

Programs: syntax

- Programs:

Programs: syntax

- Programs:

```
syntax Pgm ::= List{Decl, ";"}
```

Programs: syntax

- Programs:

```
syntax Pgm ::= List{Decl, ";"}
syntax Decl ::= FunDecl | VarDecl
```

Programs: syntax

- Programs:

```
syntax Pgm ::= List{Decl, ";"}
syntax Decl ::= FunDecl | VarDecl
syntax VarDecl ::= "int" Id
```

Programs: syntax

- Programs:

```
syntax Pgm ::= List{Decl, ";"}
syntax Decl ::= FunDecl | VarDecl
syntax VarDecl ::= "int" Id
syntax FunDecl ::= "function" Id "(" VarDecls ")" Block
```

Programs: syntax

- Programs:

```
syntax Pgm ::= List{Decl, ";"}
syntax Decl ::= FunDecl | VarDecl
syntax VarDecl ::= "int" Id
syntax FunDecl ::= "function" Id "(" VarDecls ")" Block
syntax VarDecls ::= List{VarDecl, ","}
```

Programs: syntax

- Programs:

```
syntax Pgm ::= List{Decl, ";"}
syntax Decl ::= FunDecl | VarDecl
syntax VarDecl ::= "int" Id
syntax FunDecl ::= "function" Id "(" VarDecls ")" Block
syntax VarDecls ::= List{VarDecl, ","}
```

- Other new constructs:

Programs: syntax

- Programs:

```
syntax Pgm ::= List{Decl, ";"}
syntax Decl ::= FunDecl | VarDecl
syntax VarDecl ::= "int" Id
syntax FunDecl ::= "function" Id "(" VarDecls ")" Block
syntax VarDecls ::= List{VarDecl, ","}
```

- Other new constructs:

```
syntax AExp ::= Id "(" Params ")" [strict(2)]
```

Programs: syntax

- Programs:

```
syntax Pgm ::= List{Decl, ";"}
syntax Decl ::= FunDecl | VarDecl
syntax VarDecl ::= "int" Id
syntax FunDecl ::= "function" Id "(" VarDecls ")" Block
syntax VarDecls ::= List{VarDecl, ","}
```

- Other new constructs:

```
syntax AExp ::= Id "(" Params ")" [strict(2)]
syntax Stmt ::= "return" ";"
              | "return" AExp ";" [strict]
syntax Params ::= List{AExp, ","} [strict]
```

Programs: syntax

- Programs:

```
syntax Pgm ::= List{Decl, ";"}
syntax Decl ::= FunDecl | VarDecl
syntax VarDecl ::= "int" Id
syntax FunDecl ::= "function" Id "(" VarDecls ")" Block
syntax VarDecls ::= List{VarDecl, ","}
```

- Other new constructs:

```
syntax AExp ::= Id "(" Params ")" [strict(2)]
syntax Stmt ::= "return" ";"
              | "return" AExp ";" [strict]
syntax Params ::= List{AExp, ","} [strict]
```

DEMO

Declarations

- Sequencing declarations:

  `rule D:Decl ; Ds:Pgm => D ↷ Ds`

Declarations

- Sequencing declarations:

```
rule D:Decl ; Ds:Pgm => D ↷ Ds
```

- Variable declarations:

```
rule <k> (int X:Id => . ) ...</k>
       <env> Rho:Map => Rho[X <- !L:Int] </env>
       <store> Sigma:Map => Sigma[!L <- 0] </store>
```

Declarations

- Sequencing declarations:

```
rule D:Decl ; Ds:Pgm => D ↷ Ds
```

- Variable declarations:

```
rule <k> (int X:Id => . ) ...</k>
       <env> Rho:Map => Rho[X <- !L:Int] </env>
       <store> Sigma:Map => Sigma[!L <- 0] </store>
```

- Function declarations:

```
syntax Lambda ::= "lambda" "(" VarDecls "," Block ")"
rule <k> (function X:Id ( P:VarDecls ) B => .) ...</k>
       <env> Rho:Map => Rho[X <- !L:Int] </env>
       <store> Sigma:Map => Sigma[!L <- lambda(P, B) ] </store>
```

- Demo

Utils

- The starting point of a program is the `main` function

Utils

- The starting point of a program is the `main` function
- For this, we add a marker to start the execution:

```
syntax KItem ::= "execute"
```

Utils

- The starting point of a program is the `main` function
- For this, we add a marker to start the execution:

```
syntax KItem ::= "execute"
```

- ... and a token declaration:

```
syntax Id ::= "main" [token]
```

Utils

- The starting point of a program is the `main` function
- For this, we add a marker to start the execution:

```
syntax KItem ::= "execute"
```

- ... and a token declaration:

```
syntax Id ::= "main" [token]
```

- Also, declarations are followed by `execute`:

```
<k> $PGM:Pgm ↷ execute </k>
```

Utils

- The starting point of a program is the `main` function
- For this, we add a marker to start the execution:

```
syntax KItem ::= "execute"
```

- ... and a token declaration:

```
syntax Id ::= "main" [token]
```

- Also, declarations are followed by `execute`:

```
<k> $PGM:Pgm ↷ execute </k>
```

- Now, `execute` triggers `main`:

```
rule (.Pgm ↷ execute) => main(.Params);
```

Results

- We have to refine how *results* look like:

Results

- We have to refine how *results* look like:

```
syntax KResult ::= Int | Bool
```

Results

- We have to refine how *results* look like:

```
syntax KResult ::= Int | Bool
```

vs.

```
syntax Val ::= Int | Bool
syntax Vals ::= List{Val, ","}
syntax KResult ::= Val | Vals
```

Results

- We have to refine how *results* look like:

```
syntax KResult ::= Int | Bool
```

vs.

```
syntax Val ::= Int | Bool
syntax Vals ::= List{Val, ","}
syntax KResult ::= Val | Vals
```

- Also, enable the use of `Vals` & `Val` in expressions or statements where `Params` & `AExp` (respectively) appear:

```
syntax Params ::= Vals
syntax AExp ::= Val
```

Function calls (call-by-value)

- Function call - steps:

Function calls (call-by-value)

- Function call - steps:
  1. Save the environment

Function calls (call-by-value)

- Function call - steps:
  1. Save the environment
  2. Build the new environment: declare parameters + initialise them with values

Function calls (call-by-value)

- Function call - steps:
  1. Save the environment
  2. Build the new environment: declare parameters + initialise them with values
  3. Execute the body

Function calls (call-by-value)

- Function call - steps:
  1. Save the environment
  2. Build the new environment: declare parameters + initialise them with values
  3. Execute the body
  4. Return the computed value

Function calls (call-by-value)

- Function call - steps:
  1. Save the environment
  2. Build the new environment: declare parameters + initialise them with values
  3. Execute the body
  4. Return the computed value
  5. Restore the old environment

Function calls (call-by-value)

- Function call - steps:
  1. Save the environment
  2. Build the new environment: declare parameters + initialise them with values
  3. Execute the body
  4. Return the computed value
  5. Restore the old environment
- To accomplish this we use the following 'setup' rule:

```
syntax KItem ::= "mkDecl" "(" VarDecls "," Vals ")"
syntax KItem ::= "saveEnv"
rule <k> X:Id (Vs:Vals) => saveEnv ↷ mkDecl(P, Vs) ↷ B ...</k>
     <env>... X |-> L:Int ...</env>
     <store>... L |-> lambda(P, B) ... </store>
```

Save the environment

Save the environment

Append a new cell:

```
<fstack> .List </fstack>
```

Save the environment

Append a new cell:

```
<fstack> .List </fstack>
```

Rule:

```
rule <k> saveEnv => . ...</k>
     <env> Rho </env>
     <fstack> (.List => ListItem(Rho)) ...</fstack>
```

Declare parameters + initialization

Declare parameters + initialization

Rules:

```
rule <k> mkDecl((int X:Id, Xs:VarDecls), (V:Int, Vs:Vals)) =>
                                    mkDecl(Xs, Vs) ...</k>
     <env> Rho:Map => Rho[X<-!L:Int] </env>
     <store> Sigma:Map => Sigma[!L <- V] </store>


rule mkDecl(.VarDecls,.Vals) => .
```

Execute the body

Execute the body

There is nothing to do here.

The body will be executed using the existing rules, as before.

Return + restore environment

Return + restore environment

Append a new cell to store the return value:

```
<return> .List </return>
```

Return + restore environment

Append a new cell to store the return value:

```
<return> .List </return>
```

Rules:

```
rule <k> return V:Val ; => V ...</k>
     <env> _ => Rho </env>
     <fstack> (ListItem(Rho) => .List) ...</fstack>
     <return> _ => ListItem(V) </return>
```

Return + restore environment

Append a new cell to store the return value:

```
<return> .List </return>
```

Rules:

```
rule <k> return V:Val ; => V ...</k>
     <env> _ => Rho </env>
     <fstack> (ListItem(Rho) => .List) ...</fstack>
     <return> _ => ListItem(V) </return>

syntax Val ::= "noValue"
rule <k> return ; => . ...</k>
     <env> _ => Rho </env>
     <fstack> (ListItem(Rho) => .List) ...</fstack>
     <return> _ => ListItem(noValue) </return>
```

Functions

- This implements call-by-value

Functions

- This implements call-by-value
- Exercise: call-by-reference

Data structures

Recap:

Data structures

Recap:

- Data structure: a way to organize data such that it can be accessed & modified efficiently.

Data structures

Recap:

- Data structure: a way to organize data such that it can be accessed & modified efficiently.
- Abstract datatypes: specify the operations that can be performed on a data structure & the computational complexity

Data structures

Recap:

- Data structure: a way to organize data such that it can be accessed & modified efficiently.
- Abstract datatypes: specify the operations that can be performed on a data structure & the computational complexity
- Examples: lists, arrays, trees, records, unions, classes, ...

Data structures

Recap:

- Data structure: a way to organize data such that it can be accessed & modified efficiently.
- Abstract datatypes: specify the operations that can be performed on a data structure & the computational complexity
- Examples: lists, arrays, trees, records, unions, classes, ...

In K:

Data structures

Recap:

- Data structure: a way to organize data such that it can be accessed & modified efficiently.
- Abstract datatypes: specify the operations that can be performed on a data structure & the computational complexity
- Examples: lists, arrays, trees, records, unions, classes, ...

In K:

- Autoincluded builtins: `Id`, `Int`, `Bool`, `String`, `List`, `Set`, `Map`, ...

Data structures

Recap:

- Data structure: a way to organize data such that it can be accessed & modified efficiently.
- Abstract datatypes: specify the operations that can be performed on a data structure & the computational complexity
- Examples: lists, arrays, trees, records, unions, classes, ...

In K:

- Autoincluded builtins: `Id`, `Int`, `Bool`, `String`, `List`, `Set`, `Map`, ...
- Demo: explore the `include` directory; show various functions

Next time...

... we will discuss a larger language definition with various features.

Lab this week

- Prepare for your test: solve and evaluate yourself!