Interpreter for a Functional Language

Krasimir Angelov

Chalmers University of Technology

February 25, 2010

Objective

Write an interpreter for a small, untyped functional programming language. The interpreter should walk through programs and print out the value of the main function.

The interpreter should implement either:

- *call-by-name* and *call-by-value*
- Alternative - *call-by-need*

Language Specification

- A program is a sequence of **definitions**, which are terminated by semicolons. A definition is a function name followed by a (possibly empty) list of variable names followed by the equality sign = followed by an **expression**:

$$fun\ x_1 \ldots x_n = exp\ ;$$

- An expression is one of these:

precedence	expression	example
3	identifier	foo
3	integer	512
2	application	f x
1	operation	3 + x
0	conditional	if c then a else b
0	abstraction	$\backslash x \rightarrow x + 1$

Example

```
mult x y =
    if (y < 1) then 0 else if (y < 2) then x else (x + (mult x (y-1)))

fact = \x → if (x < 3) then x else mult x (fact (x-1)) ;

main = fact 6 ;
```

Language Specification

The definition:

$$fun\ x_1 \ldots x_n = exp\ ;$$

is just syntactic sugar for:

$$fun = \backslash x_1 \rightarrow \ldots \backslash x_n \rightarrow exp\ ;$$

Run-time type checking

The language is dynamically checked i.e. during the execution you should check that:

- all variables/functions are defined
- all expressions are well-typed

Success Criteria

- The interpreter must give acceptable results for the test suite and meet the specification in the lab PM in all respects.
- All "good" programs must work with at least one of the strategies i.e. **call-by-name** or **call-by-value**.
  - The interpreter works in call-by-value by default
  - If option -n is given then it should switch to call-by-name
  - *Alternative: Implement only one strategy* - **call-by-need**
- The solution must be written in an easily readable and maintainable way.

Semantics

The semantics is defined as sequence of statements:

$$\Gamma \vdash e \Downarrow v$$

where:

- Γ - the environment i.e. mapping from function name to function definition.
- **e** - the current expression to be evaluated
- **v** - the value

Alternatively you could see this as function definition:

$$eval(\Gamma, e) = v$$

Semantics: Rules

$$\frac{\Gamma, \mathit{fun} := \mathit{body} \vdash \mathit{body} \Downarrow \mathit{value}}{\Gamma, \mathit{fun} := \mathit{body} \vdash \mathit{fun} \Downarrow \mathit{value}}$$

$$\frac{}{\Gamma \vdash \mathit{const} \Downarrow \mathit{const}}$$

$$\frac{}{\Gamma \vdash (\backslash x \rightarrow \mathit{body}) \Downarrow (\backslash x \rightarrow \mathit{body})}$$

$$\frac{\Gamma \vdash c \Downarrow i, i \neq 0 \quad \Gamma \vdash e_1 \Downarrow v}{\Gamma \vdash (\mathit{if}\ \ c\ \ \mathit{then}\ \ e_1\ \ \mathit{else}\ \ e_2) \Downarrow v} \qquad \frac{\Gamma \vdash c \Downarrow 0 \quad \Gamma \vdash e_2 \Downarrow v}{\Gamma \vdash (\mathit{if}\ \ c\ \ \mathit{then}\ \ e_1\ \ \mathit{else}\ \ e_2) \Downarrow v}$$

$$\frac{\Gamma \vdash e_1 \Downarrow v_1 \qquad \Gamma \vdash e_2 \Downarrow v_2}{\Gamma \vdash e_1 + e_2 \Downarrow v_1 + v_2}$$

Semantics: Call by name vs Call by value

- Call by value: evaluate argument before substitution

$$\frac{\Gamma \vdash \mathit{fun} \Downarrow (\backslash x \rightarrow \mathit{body}) \qquad \Gamma \vdash \mathit{arg} \Downarrow \mathit{val} \qquad \Gamma \vdash \mathit{body}[\mathit{val}/x] \Downarrow \mathit{result}}{\Gamma \vdash (\mathit{fun}\ \mathit{arg}) \Downarrow \mathit{result}}$$

- Call by name: substitute first, then evaluate

$$\frac{\Gamma \vdash \mathit{fun} \Downarrow (\backslash x \rightarrow \mathit{body}) \qquad \Gamma \vdash \mathit{body}[\mathit{arg}/x] \Downarrow \mathit{result}}{\Gamma \vdash (\mathit{fun}\ \mathit{arg}) \Downarrow \mathit{result}}$$

The problem with substitution

$$\begin{aligned}
& (\backslash x \to \backslash y \to x + y)\ 1\ 2 \\
= & ((\backslash y \to x + y)[1/x])\ 2 \\
= & (\backslash y \to (x)[1/x] + (y)[1/x])\ 2 \\
= & (\backslash y \to 1 + y)\ 2 \\
= & (1 + y)[2/y] \\
= & (1)[2/y] + (y)[2/y] \\
= & 1 + 2
\end{aligned}$$

Solution - Use Closures

We extend the set of values with a special value for closures i.e. the value is either *Integer* or *Closure*.

$$\frac{}{\Gamma \vdash (\backslash x \rightarrow body) \Downarrow Clos\ \Gamma\ (\backslash x \rightarrow body)}$$

$$\frac{\Gamma \vdash fun \Downarrow Clos\ \Delta\ (\backslash x \rightarrow body) \qquad \Delta, x := arg \vdash body \Downarrow result}{\Gamma \vdash (fun\ arg) \Downarrow result}\text{call-by-name}$$

$$\frac{\Gamma \vdash fun \Downarrow Clos\ \Delta\ (\backslash x \rightarrow body) \qquad \Gamma \vdash arg \Downarrow val \qquad \Delta, x := val \vdash body \Downarrow result}{\Gamma \vdash (fun\ arg) \Downarrow result}\text{call-by-value}$$

Recall!

$$\frac{\Gamma, fun := body \vdash body \Downarrow value}{\Gamma, fun := body \vdash fun \Downarrow value}$$

Less steps needed!

$$
\begin{aligned}
& (\backslash x \rightarrow \backslash y \rightarrow x + y)\ 1\ 2 \\
= & ((\backslash y \rightarrow x + y)[1/x])\ 2 \\
= & (x + y)[2/y, 1/x] \\
= & (x)[2/y, 1/x] + (y)[2/y, 1/x] \\
= & 1 + 2
\end{aligned}
$$

Evaluation Strategies: Comparison

- call-by-value:
  - Pros: more efficient on the current hardware
  - Cons: reasonable programs may not terminate
- call-by-name:
  - Pros: program that could terminate, terminates
  - Cons: impractical due to its computational complexity
- call-by-need:
  - Pros: combines call-by-value and call-by-name
  - Cons: sometimes difficult to reason about it

The shortest way to call-by-need

The shortest way requres destructive updates and this is what is used in practice:

1. Implement call-by-name
2. Change the implementation of the environment to contain mutable references to expressions.

Mutable References

In Haskell - use module Data.IORef or Data.STRef:

If you were using:

```
type Env = [(String, Value)]
```

replace it with:

```
type Env = [(String, IORef Value)]
```

Mutable References

- In Java, if you use java.util.ArrayList then you don't have to change anything, it is already mutable.

```
Env = java.util.ArrayList<Binding>

class Binding {
   String varName;
   Value val;
}
```

Mutable References

Modify the implementation of this two rules:

- When accessing a variable, after the evaluation, update the environment:

$$\frac{\Gamma, \mathit{fun} := \mathit{body} \vdash \mathit{body} \Downarrow \mathit{value}}{\Gamma, \mathit{fun} := \mathit{value} \vdash \mathit{fun} \Downarrow \mathit{value}}$$

- When adding a variable to the environment, in Haskell you have to create a new IORef/STRef:

$$\frac{\Gamma \vdash \mathit{fun} \Downarrow \mathit{Clos}\ \Delta\ (\backslash x \rightarrow \mathit{body}) \qquad \Delta, x := \mathit{arg} \vdash \mathit{body} \Downarrow \mathit{result}}{\Gamma \vdash (\mathit{fun}\ \mathit{arg}) \Downarrow \mathit{result}}$$

Mutable References

In pseudocode:

```
e := lookup env x
if e is evaluated
   then return e
   else e := eval e; update env x e
```

One More Problem

Look at this rule again:

$$\frac{\Gamma, \mathit{fun} := \mathit{body} \vdash \mathit{body} \Downarrow \mathit{value}}{\Gamma, \mathit{fun} := \mathit{body} \vdash \mathit{fun} \Downarrow \mathit{value}}$$

Here we lookup values by variable name which is expensive operation!

SOLUTION: de Bruijn Indices

Instead of variable names:

$$\backslash x \to \backslash y \to x + y$$

Use indexes:

$$\backslash x \to \backslash y \to \#1 + \#0$$

The index is the number of bindings from right to left!

We don't need the variable names in the environment anymore:

$$\text{type Env} = [\text{IORef Value}]$$

More About Compiling Functional Languages

Simon Peyton Jones and David Lester. Implementing Functional Languages: a tutorial. Published by Prentice Hall, 1992.

`http://research.microsoft.com/en-us/um/people/simonpj/papers/pj-lester-book/`

Consequences of Dynamic vs Static

- Advantage: Since the language is dynamically checked we could write some programs which are not possible otherwise:

$$
\begin{aligned}
\text{firefox} &= 1\ ;\\
\text{calc x y} &= \text{if firefox then x} + \text{y else plus x y}\ ;\\
\text{plus x y} &= \dots
\end{aligned}
$$

- Disadvantage: No static guarantees at all and usually less efficient.

Summary

Write an interpreter for a small, untyped functional programming language.

http://www.cse.chalmers.se/edu/course/TIN321/laborations/lab4/lab4.html