KØBENHAVNS UNIVERSITET

Assignment 4 - PLD

Mikkel Willén

31. marts 2023

Indhold

1 Domain-specific languages

a)

Below is a potential syntax for the select statement represented in PLD Lisp.

$$\begin{aligned} SelectStatement &\rightarrow (\text{Select (FROM (Where } Clause) \; Table) \; Columns) \\ Columns &\rightarrow (ColumnName) \\ Condition &\rightarrow (\text{RelOp}(Value \; Value)) \\ Value &\rightarrow \text{Constant} \end{aligned}$$

An advantages of representing the syntax as a function is that we do not need to parse the string, we can just use pattern matching instead. Another is, that the query i checked at compile time, since a syntax error, would give a syntax error og type error in the host language, and SQL injections are avoided, since the input will not be interpreted as SQL.

Advantages of representing the syntax as a data type are the same as with functions, it does however how a few limitaitions.

b)

```
(define RelOp (lambda
                        (a a)       a
                        (a b)       (<= (number? a) (number? b))))
```

c)

This is the call I would make in PLD Lisp, if I had made the functions.

```
(Select (FROM (Where < 'enrollmentYear 2021) 'students) 'studentsTables)
```

2 Limits of computation

a)

1. The recursive function definition can be syntactic, if eg. the programming language require a keyword for the function to be recursive, like rec in F#. It is also semantic, since the semantics of the programming language decides what to do with the given code, and the program must then know, if the function itself is within the scope of the function.

2. It is semantic, since the program must know and be given specific rules on how to handle unbound variables when they are called.

3. If the program uses more than 1GB of memory when running with an empty input, it has to do with how the programming language is coded and optimized, and is therefore not semantic.

4. This is semantic, since it is a matter of what the programming language does with the given code.
   Deep and shallow binding can make a difference in what the program outputs. If eg. we have a function $f1$ that binds $x = 10$ and then calls a function $f2$. $f2$ binds $x = 6$ and then calls $f3$. $f3$ then prints the variable x. When we call $f2$ with deep binding $f3$ get the environment of $f1$ and prints 10. When we call $f2$ with shallow binding $f3$ gets the environment of $f2$ and prints 6.

b)

1. The recursive function is non trivially deciable, since it is possible to create a program, that can determine whether another program contains recursive function definitions.

2. It depends on the language. For some languages it is not allowed to use an uninitialised variable, and will therefore be trivially deciable. In other languages like C, it is undecidable, since an uninitialised variable will contain the data stored in the memory it is assigned, and could therefore be anything.

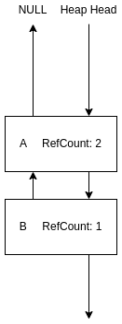
3. It is non trivially decidable, since it would be possible to check how much memory a program with no input would use. In some languages it could be trivially decidable, if all programs use more than 1GB of memory no matter what the program looks like.

4. It is non trivially deciable as long as program only contains one of the binding types. If it contains both, it would be undecidable.

3 Reference counting

a)

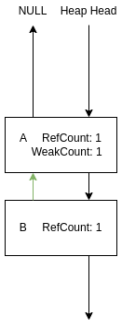
The heap head points to the first element A, which have a pointer to NULL and a pointer to B. B then points back to A and have a pointer onwards in the heap.



Figur 1: Caption

b)

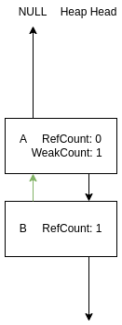
If we make the pointer from B to A a weak pointer, it changes the reference count of A. We can now free the list.



Figur 2: Caption

c)

If we free the list head, the reference count of A becomes 0, and it is now possible for us to free B.



Figur 3: Caption

4 Program transformations

a)

```
plip bingo
dingo bingo + mango 1
    let plop x = bingo x x
plop 2 + mango 3 + mango 1
bingo 2 2 + mango 3 + mango 1
    let bizarre u = u - 2 * 7
bizarre 2 + 3 * 4 - bingo 5 6 + 1 * 4 - bingo 5 6
    let bizarre u = u - 6 * 7
-12 + 12 - bizarre 5 + 4 - bizzare 5
-12 + 12 + 37 + 4 + 37
78
```

b)

```
plip' h () = (dingo' h ()) + mango' 1 ()
dingo' p () = p 2 2 + mango' 3 ()
mango' z () = z * 4 + 37
bingo' (w z) () = w - z * 7
```

If we assume these functions are in a vacuum, and no other functions exists or needs to interact with these function, we can simplify further.

```
plip' () () = 78
dingo' () () = 37
mango' z () = z * 4 + 37
bingo' (w z) () = w - z * 7
```

We can would be able to do this, since the only function we are able to call with plip is bingo, so we can compute what the function calls would be and insert that instead.

Doing lambda-lifting makes the program a lot easier to look at, since we do not how nested functions. Furthermore we can see, that a lot of the code is unnecesarry, making a lot nicer to read.

c)

```
eval (Plip' h) () = eval (Dingo' h) () + eval (Mango' 1) ()
eval (Dingo' p') () = p' 2 2 + eval (Mango' 3) ()
eval (Mango' z) () = z * 4 + 37
eval (Bingo' (w z) () = w - z * 7
```

And the second set of functions

```
eval (Plip' ()) () = 78
eval (Dingo' ()) () = 37
eval (Mango' z) () = z * 4 + 37
eval (Bingo' (w z) () = w - z * 7
```

5 Operational semantics

When looking at the rules from the book, we can see it looks a lot like while1. We use the top part of while1, with $x = 0$ instead of $x \neq 0$, and then write the rule in the bottom part.

$$\frac{\sigma(x) = 0 \vdash S_1 : \sigma \rightsquigarrow \sigma' \vdash \text{while } x \text{ do } S_1 : \sigma' \rightsquigarrow \sigma''}{\vdash \text{iterate } S_1 \text{ unless } x \; S_2 : \sigma \rightsquigarrow \sigma''}$$

We then have another rule, for when $x \neq 0$, and for this we use the top part of while2 combined with the top part of if2. We then write the rule in the bottom part.

$$\frac{\sigma(x) \neq 0 \vdash S_2 : \sigma \rightsquigarrow \sigma'}{\vdash \text{iterate } S_1 \text{unless } x \; S_2 : \sigma \rightsquigarrow \sigma'}$$

6 Parameter passing

a)

For call-by-value, the value is passed to the function, and since mango is not initialised, what happens with the function is undefined. Most likely mango would be 0, since there are a lot of zeros in memory, and we would get a div by zero. If it is not zero 57 is returned.

b)

For call-by-reference a point to the expression

```
1.0 / (float) mango
```

is passed, and we would see the same behavior as in call-by-value.

c)

Call-by-name is evaluated non-strictly and therefore is not evalutated before it is used. The function will therefore always return 57, since the argument is never used in the function.

7 Reversible languages

I have been inspired by reference 3 in chapter 12 of the course book, "principles of a reversible programming language". Specifically stack modification.
The extension to figure 12.4 is:

$$\textit{Stat} \rightarrow \text{push } \textbf{var, var}$$
$$\textit{Stat} \rightarrow \text{pop } \textbf{var, var}$$

and the extension to figure 12.5 is:

$$R(\text{push } \textbf{var, var}) = \text{pop } \textbf{var, var}$$
$$R(\text{pop } \textbf{var, var}) = \text{push } \textbf{var, var}$$

This is reversible, when the stack and the value are the same in the push and pop calls, and the program can be run both forward and backward. An example is given below:

```
read n;
f = 1;
stack = stack();
repeat
    push(f, stack);
    pop(f, stack);
    n = n - 1;
until n = 0;
write stack
```