

PLD Assignment 4

Matilde Broløs (jtw868)

March 29, 2020

The Common Part

A4C.1

The statement for adding local variables

`local var in Stat`

creates a new variable in the statement, and the variable is instantiated to 0. To get the inverse of this, we need to assure that the inverse operations are performed, to ensure that the local variable returns to the value 0. That is, the inverse of the statement is

`local var in Stat'`

where $Stat'$ is the inverse of $Stat$.

A4C.2

A gcd procedure does not output garbage in every iteration is shown below.

```
read m;
read n;

gcd()
  local g in
    if (n == 0)
      g += m;
      write g
    else
      g += m mod n
      m <-> n
      g <-> n
      gcd()
      g <-> n
      m <-> n
      g -= m mod n
    fi (n == 0)

gcd()
```

```
write m;
write n
```

This procedure first reads m and n , then calls the function `gcd()`. This function creates a local variable g and then performs the same check as the repeat-loop in the procedure from figure 13.6 in "Programming Language Design and Implementation". If n is equal to zero, this means the greatest common divisor is the value of m , and therefore g , which is currently 0, has m added to its value and is written. If n is not equal to zero, we perform the same operations as in the repeat-loop, then call `gcd()` recursively, and then performs the inverse operations to go back to the original values of the variables.

A4C.3

The gcd program written in Python looks as follows.

```
m = int(input())
n = int(input())

def swap(x,y):
    a = x
    x = y
    y = a
    return x,y

def gcd():
    global m
    global n
    g = 0
    if (n == 0):
        g += m
        print(g)
        g = 0
        assert(n==0)
    else:
        g += m % n
        m,n = swap(m,n)
        g,n = swap(g,n)
        gcd()
        g,n = swap(g,n)
        m,n = swap(m,n)
        assert(n!=0)

gcd()

print(m)
m = 0
print(n)
n = 0
```

Option 3: Type and dictionary inference using logic programming

A4O3.1

The states about types in Haskell can be written in Pure Prolog as follows.

```
isDictOf(eqInt, eq(int)).
isDictOf(ordInt, ord(int)).

isDictOf(ordBool(X), ord(bool)) :-
    isDictOf(X, ord(int)).

isDictOf(optimizedOrd, ord(prod(int, int))).

isDictOf(ordPair(X,Y), ord(prod(A,B))) :-
    isDictOf(X, ord(A)),
    isDictOf(Y, ord(B)).

isTypeOf(qsortPrime(D,L), list(A)) :-
    isDictOf(D, ord(A)),
    isTypeOf(L, list(A)).
```

I have chosen to use `isDictOf(A,B)` to represent the relationship that A is a dictionary of type B, and `isTypeOf(A,B)` to show that A is of type B.

A4O3.2

To use Prolog to find all the dictionaries that can be passed to `qsort` in the different cases, I add the following facts to implement the needed type in the program.

```
isTypeOf(list(A), list(A)).

isTypeOf(true, bool).
isTypeOf(false, bool).

isTypeOf(int, int).
```

Now let's determine the number of dictionaries that can be passed to `qsort` by running the queries seen in figure 1.

```

?- isTypeOf(qsortPrime(D,list(bool)),list(bool)).
D = ordBool(ordInt).

?- isTypeOf(qsortPrime(D,list(int)),list(int)).
D = ordInt.

?- isTypeOf(qsortPrime(D,list(prod(int,int))),list(prod(int,int))).
D = optimizedOrd ;
D = ordPair(ordInt, ordInt).

```

Figure 1: Finding dictionaries to be passed to `qsort'`.

Thus there is one dictionary for boolean lists, one for integer lists, and two for lists of pairs of integers.

Now let's look at the resolution tree for the last query. By using the *trace* command in SWI-Prolog, we can determine the way the program issues the query. This is seen in figure 2.

```

[trace] ?- isTypeOf(qsortPrime(D,list(prod(int,int))),list(prod(int,int))).
Call: (8) isTypeOf(qsortPrime(_10970, list(prod(int, int))), list(prod(int, int))) ? creep
Call: (9) isDictOf(_10970, ord(prod(int, int))) ? creep
Exit: (9) isDictOf(optimizedOrd, ord(prod(int, int))) ? creep
Call: (9) isTypeOf(list(prod(int, int)), list(prod(int, int))) ? creep
Exit: (9) isTypeOf(list(prod(int, int)), list(prod(int, int))) ? creep
Exit: (8) isTypeOf(qsortPrime(optimizedOrd, list(prod(int, int))), list(prod(int, int))) ? creep
D = optimizedOrd ;
Redo: (9) isDictOf(_10970, ord(prod(int, int))) ? creep
Call: (10) isDictOf(_11234, ord(int)) ? creep
Exit: (10) isDictOf(ordInt, ord(int)) ? creep
Call: (10) isDictOf(_11236, ord(int)) ? creep
Exit: (10) isDictOf(ordInt, ord(int)) ? creep
Exit: (9) isDictOf(ordPair(ordInt, ordInt), ord(prod(int, int))) ? creep
Call: (9) isTypeOf(list(prod(int, int)), list(prod(int, int))) ? creep
Exit: (9) isTypeOf(list(prod(int, int)), list(prod(int, int))) ? creep
Exit: (8) isTypeOf(qsortPrime(ordPair(ordInt, ordInt), list(prod(int, int))), list(prod(int, int))) ? creep
D = ordPair(ordInt, ordInt).

```

Figure 2: The result of running the query while *trace* is active.

From this the resolution tree shown in figure 3 is drawn.

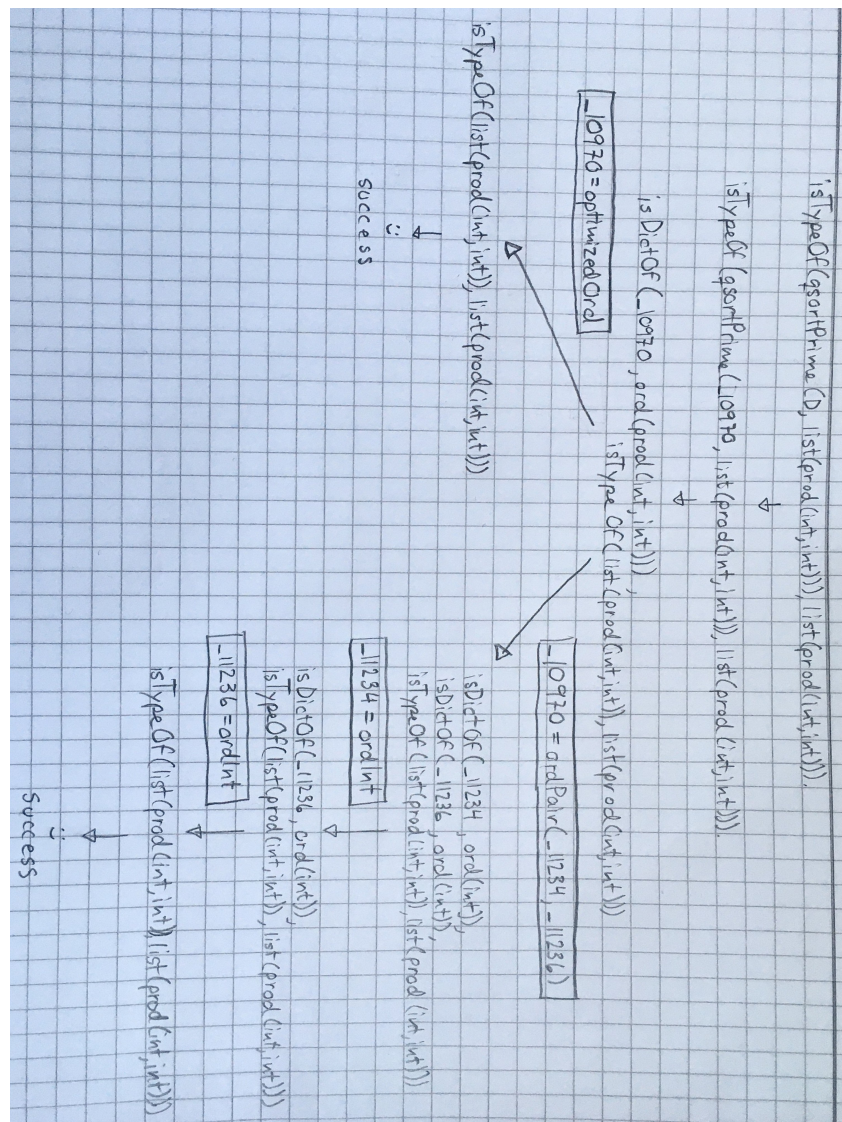


Figure 3: The resolution tree for the query.

A4O3.3

The rules for subtyping was added by the following Prolog facts and implications.

```

isTypeOf(X,B) :-
    isSubtypeOf(A,B),
    isTypeOf(X,A).
  
```

```

isSubtypeOf(bool,int).
  
```

```

isSubtypeOf(list(A),list(B)) :-
    isSubtypeOf(A,B).
  
```

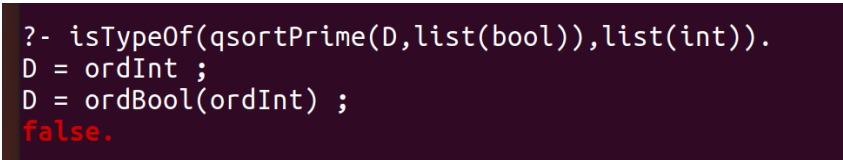
```

/*
  
```

```
isSubtypeOf(A,C) :-
    isSubtypeOf(A,B),
    isSubtypeOf(B,C).
*/
```

The last one is out commented as it produces an infinite loop, and since we will not need it we will just leave it out commented. The reason for the infinite loop is that when asking what dictionaries we can pass to `qsort` when sorting a list of booleans, outputting a list of integers, we first get one correct answer `ordInt`. Then we backtrack to try to find more solutions, and in that process we ask if `int` is a subtype of any type. We know it isn't, but because of the transitivity rule, the query will run infinitely, thus creating the infinite loop that prevents the program from terminating.

Now, in figure 4 is determined how many different dictionaries that could be passed to `qsort`, by issuing the following query.



```
?- isTypeOf(qsortPrime(D,list(bool)),list(int)).
D = ordInt ;
D = ordBool(ordInt) ;
false.
```

Figure 4: Finding dictionaries for sorting a list of booleans into a list of ints.

This means that there are two dictionaries that can be passed to `qsort`. In this case both dictionaries work perfectly fine for sorting the list, but what if we didn't know whether we had a list of booleans or a list of integers? Then it might be smartest for us to use the `ordInt` dictionary, as that dictionary can sort both integers and booleans, and thus it is the more general dictionary of the two. Thus the dictionary the compiler should choose would be `ordInt`.

A4O3.4

(a)

The following code snippet shows how **cuts** can be added to the facts and implications for O3.1 to ensure that only one dictionary of each type is found.

```
isDictOf(eqInt, eq(int)) :- !.
isDictOf(ordInt, ord(int)) :- !.

isDictOf(ordBool(X), ord(bool)) :-
    isDictOf(X, ord(int)),
    !.

isDictOf(optimizedOrd, ord(prod(int, int))) :- !.

isDictOf(ordPair(X,Y), ord(prod(A,B))) :-
    isDictOf(X, ord(A)),
    isDictOf(Y, ord(B)),
    !.
```

By adding these cuts, the query `isDictOf(X,ord(prod(int,int)))` will output only `optimizedOrd` instead of both `optimizedOrd` and `ordPair(ordInt,ordInt)`. Here I use the method of adding a cut in the end of each fact and predicate, as this will ensure that if there has been found one dictionary of a type, the query will end. This is not necessarily a very neat way to ensure this, and it results in always choosing the dictionary of a certain type which is declared first in the program. Therefore, the author should maybe put some thought into the order in which the program is written. For example, is it desirable to get the most specific dictionary, or the most general? That is something that should be considered when ordering the facts and implications.

The con of this way of choosing a specific dictionary is, that the program will choose for you, and if you are not aware that your code is ambiguous, you will never know and thus never really get to choose for yourself.

(b)

If we want to check whether there are several dictionaries of the same type, eg. `A`, we can in Prolog perform the query `isDictOf(X, A)`, and it will output all instances and conditional instances of that type - unless of course you have added cuts. Using this idea, it would be possible to check every type of dictionary manually, to determine whether multiple dictionaries of that type existed. However this would have to be done manually, or at least all types of dictionaries would have to be put in a list, so that we can loop through the list and check every type.

A4O3.5

Programming languages using type inference allows the programmer to be more free when programming, not having to worry about types all the time. This could in general make writing code easier, as it makes the procedure of writing code less complicated, and might especially be well suited for people who are new to programming, or at least to a specific programming language. Type inference makes the learning curve less steep.

It is also convenient when not quite knowing what type a function will output, maybe because there are multiple possible types of output. This lets the programmer implement one single function or procedure for something that might have needed several functions if the type wasn't inferred. In "Programming Language Design and Implementation" section 8.1 we are given a conditional as an example. Here, different branches of the conditional could give different types for the answer. Here, type inference is convenient, as we can still have one combined conditional, even though the resulting types vary.

Furthermore, in the specific case of `qsort` it is practical that the compiler can itself infer what type of dictionary it needs for sorting a list, so that the programmer need not worry about this, as it would probably not enhance the programmers understanding of the program, but would only slow the programmer down.

However, type inference can also make the code more difficult to understand, and less clear, as the programmer is not forced to have an overview of the types of the program. This often helps a programmer think his or her programs through, and thus causes the programmer to make less errors. On top of this, it also means more

work for the compiler, as it needs to perform all this type inference instead of letting the programmer do it.

And even though type inference makes the learning curve of a programming language less steep, it sometimes also prevents the programmer from truly writing good programs. Take for example C - when you first start learning it it might seem very difficult and hard to understand intuitively, but when you get good at it it might actually give you more freedom than some other programming languages.

A combination where we get some advantages of type inference, without getting the disadvantages, could be an example where the compiler infers the type whenever it is possible. Whenever types are ambiguous the compiler could produce an error, forcing the programmer to make the code unambiguous. Using this combination, we both get to be lazy, but never lets the compiler choose for us.

Another combination could be *soft typing*, as mentioned on page 154 in the notes. Soft typing never rejects programs as faulty, but instead uses dynamic type checking in the places where type inference is unambiguous. This can, according to the notes, be used as feedback for the programmer, and is often used for optimisation of dynamically typed languages.