PLD Assignment 3

Anders Lietzen Holst, wlc376

March 26, 2021

1 A3.1

Consider the directed graph $G = \langle V,\ E\rangle$ given by vertices $V = \{A,\ B,\ C,\ D\}$ and edges:

$$\begin{aligned} E = \{&(A,\ B),\ (A,\ C),\ (B,\ C),\ (B,\ D),\\ &(C,\ A),\ (C,\ D),\ (D,\ D)\} \end{aligned}$$

where $(X,\ Y)$ denotes a directed edge with source X and sink Y.

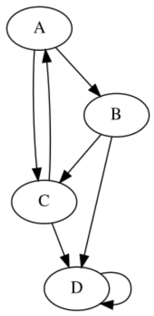
1.1 A3.1.a

- *Reproduce the graph G using Graphviz.*

I write the following Graphviz code to reproduce the graph:

```
digraph task1 {
  A -> B, C
  B -> C, D
  C -> A, D
  D -> D
}
```

Compiling this to a graph using `dot`, I get the below graph:



1.2 A3.1.b

- *Reproduce the graph G in LaTeX using a graphics package.*

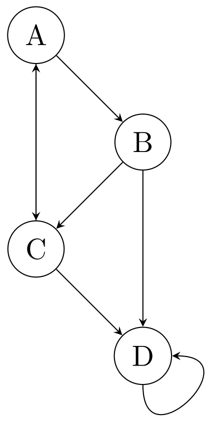
I choose Tikz, since that it the only latex package for graph drawing that I have any experience with. I write the below latex code:

```
\begin{tikzpicture}
  [v/.style={draw, circle}, % draw circles around nodes.
  node distance = 2cm,      % add some spacing between nodes.
  > = stealth               % prettier arrows.
  ]

  \node[v] (A) {A};
  \node[v] (B) [below right of = A] {B};
  \node[v] (C) [below left  of = B] {C};
  \node[v] (D) [below right of = C] {D};

  \draw[->] (A) to (B);
  \draw[->] (A) to (C);
  \draw[->] (B) to (C);
  \draw[->] (B) to (D);
  \draw[->] (C) to (A);
  \draw[->] (C) to (D);
  \draw[->] (D) to [out=270, in=0, looseness=5] (D);
\end{tikzpicture}
```

Which produces this graph:



Notice that the edge between nodes A and C is bi-directional.

1.3 A3.1.c

- *Discuss the effort of making the graph drawings using Graphviz as compared to using Tikz.*

First off, drawing the graph in Graphviz obviously took a lot less effort and lines of code than using Tikz - this is clearly evidenced by the amount of code taken to generate the two (6 lines and 17 lines for Graphviz and Tikz, respectively).

The sheer number of lines of code is, of course, not a big problem in and of itself - verbose code does not always have to be complex code. However, I identify a handful of problems with working with Tikz that are *not* problems in Graphviz.

Declaring vertices and edges

In Tikz, vertices and edges in the graph must be declared separately, and any number of edges leaving a single vertex must similarly be declared separately.

In Graphviz, two vertices `A` and `B` with a directed edge from `A` to `B` is declared simply with the syntax `A -> B`. If either of the vertices do not already exist, then it does after the declaration. Additionally, if one vertex is to have multiple outgoing edges, then these can be declared simultaneously with the notation `A -> B, C`.

It is also possible to declare vertices separately from declaring their edges. This is for example necessary if multiple vertices are to have the same name (since otherwise the name is also used for internal identification).

Positioning of elements in the graph

In Tikz, the user is not *forced* to specify positioning, but any vertices for which position is not specified will be placed on top of one another, whereas in Graphviz, vertices are positioned automatically with at least some reasonable amount of spacing if the user does not manually specify positioning.

Special edges

The graph G contains an edge from D to D. In Graphviz, this is encoded with simply `D -> D`. To properly encode this edge in Tikz, we have to specify start and end points for the edge, and we must give these in radians on the circumference of the in/out vertices.

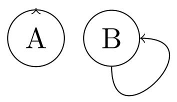
As an example, the below latex code:

```
\begin{tikzpicture}
  [v/.style={draw, circle}]

  \node[v] (A) {A};
  \node[v] (B) [right of = A] {B};

  \draw[->] (A) to (A);
  \draw[->] (B) to [out=270, in=0, looseness=5] (B);
\end{tikzpicture}
```

generates this graph:



Notice how the in- and outgoing arrows on vertex A are placed on top of another on the top of the vertex.

1.4 A3.1.d

- *With Graphviz, adding a single edge to a graph can in some cases significantly alter the layout of that graph. Discuss advantages and disadvantages of this.*

When drawing graphs, Graphviz will always attempt to draw the graph such that there are no edges intersecting. This is, of course, only possible when the graph is planar, but often even a planar graph must be carefully rearranged to be drawn on a 2D surface with no intersections.

Even if adding an edge to a planar graph results in a planar graph, it is often necessary to rearrange the graph to avoid intersections, and potentially even to the point of the graph becoming unrecognizable. As such, I would argue that it is very permissible for Graphviz to sometimes drastically alter a graph when an edge is added.

Additionally, it is often very convenient to not have to position vertices manually when drawing a graph. This becomes increasingly helpful when adding graphs to already existing, large graphs.

However, drawing a graph with intersections can sometimes be preferable to rearranging it even if the graph is planar - for example if some features of the graph are only visually obvious when the graph is arranged in a certain way. In such cases it can be an enormous annoyance to not control over vertex placement - fortunately, however, Graphviz (like Tikz and basically every other graph drawing utility out there) supports manual positioning of vertices.

---

2 A3.2

Consider the below function written in SML

```
fun dingo (x, nil)    = false
  | dingo (x, y::ys) = x = y orelse dingo (x, ys);
```

2.1 A3.2.a

- *Explain what the function does.*

Given a list `ys` and an element `x`, a call to `dingo (x, ys)` returns `true` if `x` exists in `ys`; else `false`.

This is because: The first function clause states that if `ys` is `nil` (the empty list), then surely `x` is not a member of it, whereas the second function clause returns `true` if `x` is equal to the head of the list `ys`; else a recursive call to examine the tail of the list is made.

2.2 A3.2.b

- *Use the Hindley-Milner algorithm to infer the type of* `dingo`*.*

I first initialize a type environment ρ_0 (corresponding to the first function clause) with type variables for the first function clause of `dingo` with parameters `x` and `nil`, as well as an empty set of global type bindings σ:

$$\rho_0 = [\texttt{dingo} \mapsto ((A * B) \rightarrow C), x \mapsto A, \mathit{nil} \mapsto B]$$
$$\sigma = [\,]$$

Now, I wish to infer the type of the function body in the new type environment ρ_0.

I first wish to unify B with the actual type of `nil`. Since `nil` is a null list literal, I know its type to be list D for some new type variable D, and I must then compute unify(B, list D).

B is an unbound type variable, so the unification is successful if list D does not contain B. The occurs check passes since D does *not* contain B; thus unification is successful, and I add the binding $B \mapsto$ list D to σ:

$$\sigma := \sigma[B \mapsto \text{list } D]$$

I also infer this function clause to have type `bool`, since the return value `false` is a boolean literal. Since in ρ_0, the return type of `dingo` is the type variable C, I now need to compute unify(C, `bool`).

Since C is an unbound type variable, the unification is successful if C does not contain `bool`, but this is trivially true since `bool` is a type constructor with zero parameters. Thus I add the binding ($C \mapsto$ `bool`) to σ:

$$\sigma := \sigma[C \mapsto \texttt{bool}]$$

Now, I want to infer the type of the second function clause given the bindings acquired in the first. I continue with the type environment ρ_0, and at this point, the set of global bindings σ is:

$$\sigma = [B \mapsto \text{list } D,\ C \mapsto \texttt{bool}]$$

Consider now the second function clause. Since the second parameter is matched against a head/tail list pattern, I should add `y` to ρ_0, creating a new type environment ρ_1 given by:

$$\rho_1 := \rho_0[\texttt{y} \mapsto D]$$

Since `ys` maps to list D.

I now want to infer the type of function clause body. Since it consists simply of a logical OR expression, I know that the type of the body is `bool`. Unifying C (the return type of `dingo`) with `bool` is successful since there exists a mapping from C to `bool` in σ.

However, to successfully infer the type of the function body, the logical OR expression must also be well-formed - it is well-formed if both of its operands are of `bool` type.

I start with the left-hand operand `x = y`. This equality is well-typed if its operands are of the same type. To find this out, I compute the unification of the types of `x` and `y` which I have given in the local type environment:

$$\text{unify}(\rho(\texttt{x}),\ \rho(\texttt{y})) = \text{unify}(A,\ D)$$

The unification is determined using unification rule 4 (as given in the lecture notes) - this holds because A is an unbound type variable and D does not contain A. D is shown to not contain A by the fact that D is itself an unbound type variable different from A.

As such unify(A, D) holds - a new type variable E is added to the set of global bindings σ:

$$\sigma := \sigma[D \mapsto A]$$

This also updates the mapping of B to list A.

Since unification of `x` and `y` was successful, we only need the second operand of the logical OR expression to be well-typed. This is the case if it is also a `bool` type. Since it is a recursive call to `dingo`, which we know from the local type environment to have return type `bool`, this also holds. As such unification of the two function clauses is successful.

The entire function has been type inferred, and the resulting set of type bindings is:

$$\sigma = [B \mapsto \text{list } A, C \mapsto \texttt{bool}, D \mapsto A]$$

The type of the function is then:

$$\texttt{dingo} \ : \ (A * \text{list } A) \to \texttt{bool}$$

By replacing unbound type variables (there is only A) with type parameters, the type is generalized and its type schema is then:

$$\texttt{dingo} \ : \ \forall\alpha \ . \ (\alpha * \text{list } \alpha) \to \texttt{bool}$$

3 A3.3

- *Extend the list program with a predicate `longerthan/2` such that `longerthan(L1, L2)` holds if the list `L1` is longer than the list `L2`.*

The assignment text does not state what the predicate should return if either `L1` or `L2` is not a well-formed list. For example, what should be the value of the below statement:

```
longerthan(cons(3, cons(4, foo)), cons(2, nil)).
```

Clearly the first argument is not a well-formed list, but a naive implementation might simply determine that the first list has more layers of "cons" and thus conclude that it must be the longer list.

However, I will assume that the predicate should be `false` when either one or both of the lists are not well-formed lists. Thus my solution is:

```
% if both L1 and L2 have a tail, then L1 is longer than T2 if T1's tail is
% longer than T2's tail.
longerthan(cons(_, T1), cons(_, T2)) :- longerthan(T1, T2).

% if L1 has a tail and L2 is nil, then L1 is longer than L2. however, we must
% also assert that L1 is, in fact, a well-formed list.
longerthan(cons(_, T1), nil) :- list(T1).
```

Where `list/1` is as defined in the lecture slides.

4 A3.4 - resubmission

- *Extend the operational semantics of the imperative language (as presented in the lectures) with transition rules for the* `repeat` *loop statement.*

~~`repeat s until x` iterates its loop body `s` until the condition `x` becomes non-zero – as such, it is equivalent to a `do while (!x)` loop, and its loop body is executed at least once.~~

In the feedback to my hand-in, I was told that I had mistaken the semantics of the `repeat` loop. My understanding was that the loop body `s` would execute until `x` became *non-zero*, but `s` is actually executed until `s` is *non-zero*.

This would imply that a `repeat` loop always modifies the store σ. Consider the below transition rule for the case where `x` is ~~non-zero~~ zero:

$$\frac{\sigma \vdash \texttt{s} \rightsquigarrow \sigma' \quad \cancel{\sigma''(\mathrm{x}) \neq 0} \quad \sigma''(\mathrm{x}) = 0}{\sigma \vdash \texttt{repeat s until x} \rightsquigarrow \sigma'}$$

The transition rule states that:

- if executing the loop body `s` modifies the current store σ to some modified store σ', and
- if under σ', the loop condition `x` evaluates to ~~a non-zero value~~ zero,
- then the entire loop statement results in a transition from σ to σ' .

In other words, the store is only modified by the first (guaranteed) iteration. Consider now the transition rule for the case where `x` is ~~zero~~ non-zero:

$$\frac{\sigma \vdash \texttt{s} \rightsquigarrow \sigma'' \quad \cancel{\sigma''(\mathrm{x}) = 0} \quad \sigma''(\mathrm{x}) \neq 0 \quad \sigma'' \vdash \texttt{repeat s until x} \rightsquigarrow \sigma'}{\sigma \vdash \texttt{repeat s until x} \rightsquigarrow \sigma'}$$

This rule states that:

- if executing the loop body `s` modifies the current store σ to some modified store σ'', and
- if under σ'', the loop condition `x` evaluates to ~~zero~~ a non-zero value, and
- if under σ'', the loop statement modifies σ'' to some new, modified store σ',
- then executing the entire loop statement results in a transition from σ to σ'.

Note that the loop condition `x` *is evaluated under the store* σ'' *arrived at after the first iteration.* This is important! ;)

---

5 A3.5

- *Extend the type rules of the functional language (as presented in the lectures) with rules for the new* ***`private .. within`*** *expression.*

I extend the type rules with the below rule:

$$\frac{\begin{array}{r}\tau \vdash e_1 \;:\; t_1 \\ \tau[x \mapsto t_1] \vdash e_2 \;:\; t_2\end{array}}{\tau \vdash \texttt{private}\ x = e_1\ \texttt{within}\ e_2 \;:\; t_2}$$

The rule states that:

- if e_1 has the type t_1 in the type environment τ, and
- if, after extending τ with the type binding $x : t_1$, the expression e_2 has type t_2,
- then `private` $x = e_1$ `within` e_2 has type t_2

Interestingly, the `private .. within` expression is simply a renaming of the more well-known `let .. in` expression from the ML family of languages.