# TLC Spring 2023 - Graph Invariants with Dependent Types

Yiwei Wu Raphael Sofaer

12 May 2023

## 1 Introduction

Graphs are a common data structure in both practical Software Engineering and in Computer Science. This abstraction consists of a set of nodes and a set of edges linking those nodes. Many algorithms exist for extracting data from graphs, constructing paths in graphs, dividing graphs into subgraphs, find max-flows through graphs, and more. Many real-world data sets such as cartographic maps, social networks, academic citations, and program control flow can be represented as graphs. These practical graphs often attach complex data in the form of annotations to nodes and edges, and have delicate structures which must be understood by a programmer who is extracting data.

Unfortunately, while the basic graph data structure and many elaborations of it such as hypergraphs and multigraphs have been explored in detail, we still lack the tools to enforce constraints on the graph structure specific to a given application. At best, extracting data from a complex graph format depends on informal documentation and experimentation.

We propose and demonstrate a novel technique for the enforcement of local and global structural graph invariants using dependent types, and provide a demonstration of this technique using an adjacency list graph implementation.

## 2 Background

Any program which extracts data from a graph via traversal depends on the graph having a particular structure. Consider a graph of user-network interactions on a telecom network [Hua(2023)]. This graph consists of four node types: user, package, app, and cell, and 3 edge types: user-buy-package, user-use-app, and user-live-cell. One seemingly simple question may be, is a user who buys a certain package more likely to have a certain type of cell phone? We can traverse from package nodes to user nodes, then to cell nodes, in order to extract tabular data on which we can calculate our desired statistics. However, in order to write that code we must know whether a user who buys a package is guaranteed to have used a cell phone. Even if in the data file we are currently considering, all users nodes are connected to cell nodes, that may not be the case in the next data set we receive.

Dependent types provide a possible solution to this problem. A function extracting data from a graph may be written to operate on a graph type augmented with the structural invariants that the function depends on. Since a dependent type system allows any term to be made a condition of a type, any invariant may be incorporated into a type-level constraint. In the above case, the code providing the graph would define a type which incorporates the invariant that ensures the traversals made in the functions are valid. In this case, the invariant would ensure that every user with a user-buy-package edge has a user-live-cell edge.

## 3 Implementation

### 3.1 Pi-Forall

We implement our proposal using Pi-ForAll, a learning language implemented as only a type checker by Professor Stephanie Weirich [Weirich(2022)]. Pi-Forall consists of a dependent type checker, but no evaluator. We implement the Sigma types described in the Pi-ForAll paper (already present in the `full` reference implementation), and implement a constrained graph type.

### 3.2 Added Sigma Type Rules

We implement the sigma rules in the Pi-ForAll paper in bidirectional style in the `version1/src/TypeCheck.hs` file [Sofaer and Wu(2023)]. To avoid any errors, we use the provided type checker in `full/src/TypeCheck.hs` to typecheck the graph types describe below.

### 3.3 Design

A constrained graph type must contain 3 pieces: First, the graph itself. Second, the constraint. Finally, a proof that the constraint is met. In addition, because we use an n-sized vector type for our graph, we have a fourth piece of data, the number of nodes n. Since the proof must be able to refer to all previous pieces of the type, pi-forall requires us to use an indexed sigma type.

```
data Graph (n : Nat) : Type where
  G of (Vec (List Node) n)

GraphConstraint : Type
GraphConstraint = [n: Nat]
    -> Graph n -> Bool

ConstrainedGraph : Type
ConstrainedGraph = { n : Nat | -- size
    { g: Graph n |
        { con : GraphConstraint |
            con [n] g = True -- proof
    }}}
```

### 3.4 Proof

We provide a proof confirming that the type above requires the constraint to evaluate to `True` for the graph it is paired with. We omit the definitions of `con_graph_con`, `con_graph_graph` and `con_graph_size`, which extract the pieces of the `ConstrainedGraph` sigma type. For these definitions, please refer to Graph.pi

```
con_graph_proof:(cg: ConstrainedGraph)
    -> (con_graph_con cg)
         [con_graph_size cg]
         (con_graph_graph cg) = True
con_graph_proof = \cg .
    let (n, wrapped_g) = cg in
        let (g, con') = wrapped_g in
            let (con, proof) = con' in
                proof
```

### 3.5 Examples

To test this definition, we constructed three examples of growing complexity. First, we constrain a graph to have at least one node. Second, we constrain a graph to have all nodes in the graph have degree at least 1. Third, we constrain a graph to have no self-edges. Finally, we constrain a graph to be in topological order, and therefore be acyclic. See Appendix 1 5 for the code, or see the code on Github [Sofaer and Wu(2023)].

## 4 Conclusion

In this paper we describe a method for using dependent types to construct and enforce arbitrary local or global constraints on graphs. We provide examples of using this method for simple local and global graph invariants. To the best of our knowledge, no open source libraries exist implementing graph constraints using dependent types[1].

Further work, if pursued, would re-implement the types in a practical dependently typed language such as Idris, and provide matching graph algorithms and constrained types. An acceptable open source graph library would support directed and undirected graphs with annotations on both nodes and edges. In addition, it would provide an interface for defining and combining constraints in a manner amenable to type checking.

[1] A thesis topic was started in 2018 on the subject but no published work was found [fun(2023)].

# References


[Hua(2023)] 2023. SNAP: Telecom Graph. https://snap.stanford.edu/data/telecom-graph.html [Online; accessed 11. May 2023].

[fun(2023)] 2023. Universität Tübingen - Implementing functional graph algorithms in a dependently typed language. https://ps.informatik.uni-tuebingen.de/teaching/thesis/2018/05/30/idris-graph-algos [Online; accessed 11. May 2023].

[Sofaer and Wu(2023)] Raphael Sofaer and Yiwei Wu. 2023. pi-forall with constrained graphs. https://github.com/rsofaer/pi-forall/blob/tlc_spring_2023/full/pi/Graph.pi [Online; accessed 11. May 2023].

[Weirich(2022)] Stephanie Weirich. 2022. Implementing Dependent Types in pi-forall. *arXiv* (July 2022). https://doi.org/10.48550/arXiv.2207.02129 arXiv:2207.02129

# 5 Appendix 1: Example Graph Code

```
hasNodeCon : GraphConstraint
hasNodeCon = \[n] g . case g of
  G v -> case v of
    Nil -> False
    Cons [a] h tl -> True

cg_simplest : ConstrainedGraph
cg_simplest = (1, (G (Cons [0] (Nil) Nil), (hasNodeCon, Refl)))
cg_simplest_fail : ConstrainedGraph
cg_simplest_fail = (0, (G (Nil), (hasNodeCon, Refl)))

degreeOneCon' : [n: Nat] -> Vec (List Node) n -> Bool
degreeOneCon' = \[n] v . case v of
  Nil -> True
  Cons [m] h tl -> case h of
    Nil -> False
    Cons lst_h lst_tl -> degreeOneCon' [m] tl
degreeOneCon : GraphConstraint
degreeOneCon = \[n] g . case g of
    G v -> degreeOneCon' [n] v

cg_degOne: ConstrainedGraph
cg_degOne= (1, (G (Cons [0] (Cons 0 Nil) Nil), (degreeOneCon, Refl)))
cg_degOneEmpty: ConstrainedGraph
cg_degOneEmpty = (0, (G (Nil), (degreeOneCon, Refl)))
cg_degOneFail : ConstrainedGraph
cg_degOneFail = (1, (G (Cons [0] (Nil) Nil), (degreeOneCon, Refl)))

noSelfEdgeConstraint' : [n: Nat] -> Vec (List Node) n -> Nat -> Bool
noSelfEdgeConstraint' = \[n] v idx . case v of
  Nil -> True
  Cons [m] h tl -> if contains_nat h idx then False
                   else noSelfEdgeConstraint' [m] tl (Succ idx)

noSelfEdgeConstraint : GraphConstraint
noSelfEdgeConstraint = \[n] g . case g of
  G v -> noSelfEdgeConstraint' [n] v Zero

cg_noSelfEdge : ConstrainedGraph
cg_noSelfEdge = (2, (G (Cons [1] (Cons 1 Nil) (Cons [0] (Cons 0 Nil) Nil)),
    (noSelfEdgeConstraint, Refl)))

cg_selfEdgeFail : ConstrainedGraph
cg_selfEdgeFail = (1, (G (Cons [0] (Cons 0 Nil) Nil),
    (noSelfEdgeConstraint, Refl)))

-- Topologically sorted and therefore acyclic
acyclicConstraintSingle: [n: Nat] ->  Nat -> List Node -> Bool
acyclicConstraintSingle = \[n] idx lst . case lst of
  Nil -> True
  Cons h tl -> case gt h idx of
    True -> acyclicConstraintSingle [n] idx tl
```

```
    False -> False

acyclicConstraint' : [n: Nat] ->  Vec (List Node) n -> (idx: Nat) -> Bool
acyclicConstraint' = \[n] v idx . case v of
  Nil -> True
  Cons [m] h tl -> if acyclicConstraintSingle [n] idx h
                   then acyclicConstraint' [m] tl (Succ idx)
                   else False

acyclicConstraint : GraphConstraint
acyclicConstraint = \[n] g . case g of
  G v -> acyclicConstraint' [n] v Zero

-- Example of an acyclic Graph
cg_acyclic: ConstrainedGraph
cg_acyclic = (2, (G (Cons [1] (Cons 1 Nil) (Cons [0] (Nil) Nil)),
    (acyclicConstraint, Refl)))

-- Example of a cyclic graph
cg_cyclic: ConstrainedGraph
cg_cyclic = (2, (G (Cons [1] (Cons 0 Nil) (Cons [0] (Cons 1 Nil) Nil)),
    (acyclicConstraint, Refl)))
```