A Domain-Specific Programming Language for Robots based on Instruction Graphs

Andrew Benson[*]

Computer Science Department

Carnegie Mellon University

Pittsburgh, USA

adbenson@andrew.cmu.edu

ABSTRACT

Instruction graphs are a recently proposed data structure that encodes a sequence of actions for a robot to perform [1]. For example, in order for a robot to purchase coffee, a robot might move forward, looking for a barista every 5 meters, then ask for a coffee once a barista is found. An instruction graph is an explicit representation of these actions, where each action is a vertex and edges are the transitions between them. We contribute a formalization of instruction graphs as a domain-specific programming language, prove type safety for the language, and provide an interpreter.

[*]Only the student author is listed here for this written report. The author also wishes to credit Jonathan Aldrich, Joshua Sunshine, and Cyrus Omar for their contributions.

WOODSTOCK '97 El Paso, Texas USA

© 2016 ACM. ISBN 123-4567-24-567/08/06...$15.00

DOI: 10.475/123_4

CCS Concepts

•**Computer systems organization** → *Robotic control;* •**Software and its engineering** → *Domain specific languages;*

Keywords

Instruction Graphs; Robotic Domain Specific Languages

1. INSTRUCTION GRAPHS

Being able to follow sequences of actions is a very important function of a robot. Many tasks that a robot is programmed to do are already in the form of a sequence of actions. For example, there is a very natural sequence of actions a robot might do in order to navigate a maze - turn left, move forward five meters, turn right, etc. Some tasks may also involve conditional actions, so that the robot may branch in its execution of actions dependent on some environmental condition. For example, instead of traveling a predetermined five meters forward, a robot might move forward until it senses a wall a meter away. We are interested in this set of tasks that can be expressed as a composition of primitive robotic actions.

Instruction graphs are a recently proposed data structure by which such tasks can be encoded and expressed [1]. Informally, an instruction graph consists of vertices, which can be thought of as actions a robot might do like “move forward five meters” or conditions a robot may check like “is there a wall one meter away”, and edges, which connect the actions in a sequence or in a branching pattern (in the case of a conditional vertex). Instruction graphs distinguish four classes of vertices: **Do** vertices, which represent a single action executed once; **DoUntil** vertices, which represent an action executed until a condition is met; **Conditional** vertices, which represent a condition to be checked to determine which vertex to proceed to; and **GoTo** vertices, which merely represent an advancement to another vertex and are usually used only to implement loops.

Instruction graphs have already been detailed and used in other research. In [1], instruction graphs are generated by a robot by parsing natural language whenever a user verbally specifies a task to the robot. However, most of the attention was focused on the spoken interaction between the human and robot and how the instruction graph is generated. Instruction graphs were an internal detail of the robot and could not be created unless generated via spoken interaction with the robot.

2. APPROACH

Our goal in this research is to explore the capabilities of instruction graphs and to formalize their creation and execution. We have three main contributions to make in this direction:

- Formalization of instruction graphs as a domain-specific programming language - the instruction graph language
- Proof of type safety for this language
- Interpreter for the instruction graph language on the Turtlebot simulator

For the first point, we formalize instruction graphs into a programming language by giving a grammar, specifying what it means to be well-formed by detailing statics, and specifying its execution through a description of its dynamics.

For the second point, we first define theorems of progress and preservation for the language, then prove them to ensure that the language is safe.

For the third point, we write a lexer and parser to read concrete programs of the language, check statics on the program, and interpret the program as specified by the dynamics. We demonstrate the feasibility of the

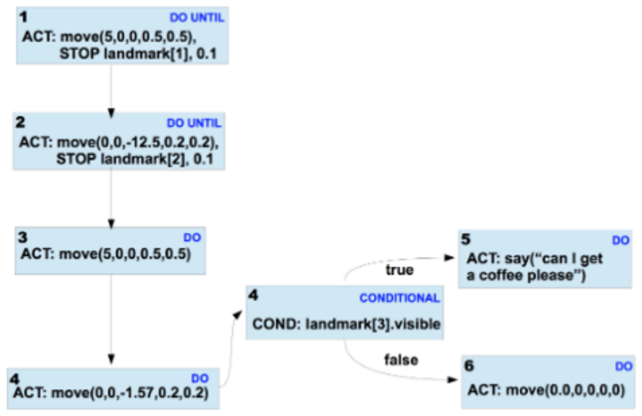
1
DO UNTIL
ACT: move(5,0,0,0.5,0.5),
STOP landmark[1], 0.1
2
DO UNTIL
ACT: move(0,0,-12.5,0.2,0.2),
STOP landmark[2], 0.1
3
DO
ACT: move(5,0,0,0.5,0.5)
4
DO
ACT: move(0,0,-1.57,0.2,0.2)
4
CONDITIONAL
COND: landmark[3].visible
true
false
5
DO
ACT: say("can I get a coffee please")
6
DO
ACT: move(0.0,0,0,0,0)

Figure 1: The instruction graph for the "get coffee" program.

language by simulating the execution of the program in the Turtlebot simulator.

3. INSTRUCTION GRAPH LANGUAGE

Before formally specifying the language via its grammar, statics, and dynamics, it is helpful to see an example of an instruction graph and its corresponding program in the instruction graph language.

3.1 The "Get Coffee" Program

In Figure 1 we see the instruction graph representing a robot's task to get coffee. Admittedly, the task itself is a bit contrived. The robot begins by moving with parameters (5, 0, 0, 0.5, 0.5) (the meaning of the parameters is unknown without knowing more about the robot, but one could guess that it specifies at least speed, distance, and rotation) until it is within 0.1 meters of landmark[1], then changes direction and moves with parameters (0, 0, -12.5, 0.2, 0.2) until it is within

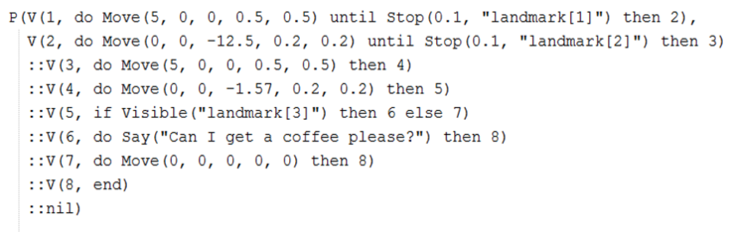
```
P(V(1, do Move(5, 0, 0, 0.5, 0.5) until Stop(0.1, "landmark[1]") then 2),
  V(2, do Move(0, 0, -12.5, 0.2, 0.2) until Stop(0.1, "landmark[2]") then 3)
  ::V(3, do Move(5, 0, 0, 0.5, 0.5) then 4)
  ::V(4, do Move(0, 0, -1.57, 0.2, 0.2) then 5)
  ::V(5, if Visible("landmark[3]") then 6 else 7)
  ::V(6, do Say("Can I get a coffee please?") then 8)
  ::V(7, do Move(0, 0, 0, 0, 0) then 8)
  ::V(8, end)
  ::nil)
```

Figure 2: The corresponding instruction graph program for the "get coffee" program.

0.1 meters of landmark[2]. Then it moves with parameters (5, 0, 0, 0.5, 0.5) and then with parameters (0, 0, -1.57, 0.2, 0.2). Following that, it checks whether landmark[3] is visible, and if so, it says "can I get a coffee please", otherwise moves with parameters (0, 0, 0, 0, 0).

With this instruction graph we can see most of the features present in instruction graphs - **Do** vertices, **DoUntil** vertices, **Conditional** vertices, move actions, condition checking, etc. We miss seeing **GoTo** vertices, but those are very straightforward - the robot simply moves to the next vertex.

In Figure 2 we see the corresponding instruction graph program for getting coffee. The similarity between the instruction graph and the instruction graph program should be quite clear.

3.2 Discussion of Grammar

As can be seen from inspecting the structure of the example program above, programs written in the instruction graph language are quite simple and closely follow the layout of an instruction graph. Informally, programs consist of a start vertex and a list of all other vertices in the program, all of which are given an index

`Program`	p	$::=$	$\mathbf{P}(v,\ vs)$	programs
`Vertices`	vs	$::=$	nil	empty
			v **::** vs	cons
`Vertex`	v	$::=$	$\mathbf{V}(n,\ c)$	vertex
`Content`	c	$::=$	**do** a **then** n	single action
			do a **until** cnd **then** n	open loop action
			if cnd **then** n **else** n	conditional
			goto n	goto
			end	termination

Figure 3: Grammar for the instruction graph language.

to identify them. Each vertex is one of the classes discussed earlier - **Do**, **DoUntil**, **Conditional**, or **GoTo** - and contains the appropriate content (action or condition or neither) for its class. The exception is **End**, which signals that this vertex is the last vertex to be executed in the program. Each vertex also lists the possible vertices that could follow it during execution.

Figure 3 shows the grammar for the instruction graph language. Note that a few parts of the language have been omitted from this grammar. Firstly, note that $n \in \mathbb{Z}$, the integers. Secondly, since the actions and conditions that a particular robot could respectively do or check varies by the robot, a grammar for them has been omitted. The actions a and conditions c still need to be well-specified by the implementor, though. For instance, for the particular robot the "get coffee" program above was written for, actions a can be either Move, which takes in five floats as arguments, or Say, which takes a string as an argument.

It is worth noting that the instruction graph language is unlike most programming languages. Programs in this language are not traditional expressions that can be evaluated to model a mathematical function, as in most languages, but are descriptions of graphs. This language also has no types or values, at least in the traditional sense, but has many side-effects, which is opposite most programming languages. It is worth emphasizing that the instruction graph language is a *domain specific language* for specifying tasks to robots, not a general purpose programming language, so the language should be expected to look and behave differently from conventional programming languages.

4. STATICS

Most programming languages define statics by giving rules for when expressions in the language are well-typed. Since the instruction graph language has no types, the statics are a bit trickier. Fundamentally,

$$\frac{(\mathbf{V}(s,\ c) :: vs,\ U)\ \texttt{defined} \qquad (\mathbf{V}(s,\ c) :: vs,\ \emptyset,\ s,\ U)\ \texttt{connected}}{\mathbf{P}(\mathbf{V}(s,\ c),\ vs)\ \texttt{valid}}(\texttt{P}_{\texttt{valid}})$$

Figure 4: Statics: valid judgment.

$$\frac{}{(nil,\ \{\ \})\ \texttt{defined}}(\texttt{nil}_{\texttt{defined}})$$

$$\frac{(vs,\ U)\ \texttt{defined} \qquad n \notin U}{(\mathbf{V}(n,\ c) :: vs,\ U \cup \{n\})\ \texttt{defined}}(\texttt{cons}_{\texttt{defined}})$$

Figure 5: Statics: defined judgment.

$$\frac{(vs,\ U)\ \texttt{defined} \qquad U_v \subseteq U \qquad n \in U_v}{(vs,\ U_v,\ n,\ \emptyset)\ \texttt{connected}}(\texttt{visited}_{\texttt{connected}})$$

$$\frac{(vs,\ U)\ \texttt{defined} \qquad U_v \subseteq U \qquad \mathbf{V}(n,\ \mathbf{end}) \in vs \qquad n \notin U_v}{(vs,\ U_v,\ n,\ \{n\})\ \texttt{connected}}(\texttt{end}_{\texttt{connected}})$$

$$\frac{\begin{array}{c}\mathbf{V}(n,\ \mathbf{do}\ a\ \mathbf{then}\ n') \in vs \\ (vs,\ U_v \cup \{n\},\ n',\ U)\ \texttt{connected} \qquad n \notin U_v\end{array}}{(vs,\ U_v,\ n,\ U \cup \{n\})\ \texttt{connected}}(\texttt{doonce}_{\texttt{connected}})$$

$$\frac{\begin{array}{c}\mathbf{V}(n,\ \mathbf{do}\ a\ \mathbf{until}\ cnd\ \mathbf{then}\ n') \in vs \\ (vs,\ U_v \cup \{n\},\ n',\ U)\ \texttt{connected} \qquad n \notin U_v\end{array}}{(vs,\ U_v,\ n,\ U \cup \{n\})\ \texttt{connected}}(\texttt{dountil}_{\texttt{connected}})$$

$$\frac{\begin{array}{c}\mathbf{V}(n,\ \mathbf{if}\ cnd\ \mathbf{then}\ n'\ \mathbf{else}\ n'') \in vs \qquad (vs,\ U_v \cup \{n\},\ n',\ U)\ \texttt{connected} \\ (vs,\ U_v \cup U \cup \{n\},\ n'',\ U')\ \texttt{connected} \qquad n \notin U_v\end{array}}{(vs,\ U_v,\ n,\ U \cup U' \cup \{n\})\ \texttt{connected}}(\texttt{ifthen}_{\texttt{connected}})$$

$$\frac{\mathbf{V}(n,\ \mathbf{goto}\ n') \in vs \qquad (vs,\ U_v \cup \{n\},\ n',\ U)\ \texttt{connected} \qquad n \notin U_v}{(vs,\ U_v,\ n,\ U \cup \{n\})\ \texttt{connected}}(\texttt{goto}_{\texttt{connected}})$$

Figure 6: Statics: connected judgment.

though, statics defined what it means for a program in the language to be well-formed. Thus we can list several desirable characteristics for programs of this language:

- No two vertices should have the same index.
- Every vertex that follows another vertex should be defined.
- There should exist a path from the start vertex to every defined vertex.

Since programs can be thought of as descriptions of graphs, these characteristics make intuitive sense when considering them as so. The first item thus states that no vertex is defined twice. The second says that every edge should connect to a defined vertex. The third expresses the idea that the instruction graph that the program represents should be a connected graph.

When all of these conditions are met, we consider the program to be a valid one. We formally judge a program to be valid using the judgment shown in Figure 4. In other words, a program is valid when it defines a set of vertex indices U, and the start vertex is connected to exactly the vertices in U.

Of course, we must define what it means to be defined and connected as well. Those are formally shown in Figures 5 and 6.

Informally, a program defines a set U of vertex indices if its list of vertices only defines each vertex once and each vertex index in the list appears in U and vice versa.

The connected judgment is more tricky. The idea is to do a BFS search from the start vertex outward, keeping track of which vertices have been visited, and discovering which vertices are connected that haven't been visited.

The appendix reproduces the statics and explains them in more detail as well.

5. DYNAMICS

To write dynamics for the instruction graph language we need a machine model for how programs in the language are evaluated and executed. Intuitively, this is clear - begin at the vertex representing the start vertex, execute its action, and continue as directed until we hit **End**. But to be precise and rigorous, we have to also keep track of both the inputs to the program i.e. whether the conditions the robot checks are true and the outputs of the program i.e. the actions executed.

To that end we define a *configuration* which is a 4-tuple that describes the state of a program midway through execution. It consists of the index n of the current vertex being considered, the list vs of all vertices in the program, the list I of inputs to the program, and the list O of outputs of the program.

A configuration is defined formally in Figure 7.

$$\texttt{Configuration} \quad cfg \quad ::= \quad (n,\ vs,\ I,\ O) \qquad \text{configurations}$$

Figure 7: Dynamics: configuration definition.

$$\frac{\mathbf{V}(n,\ \mathbf{end}) \in vs}{(n,\ vs,\ I,\ O)\ \texttt{terminated}}(\texttt{cfg}_{\texttt{terminated}})$$

Figure 8: Dynamics: terminated judgment.

$$\frac{\mathbf{V}(n,\ \mathbf{do}\ a\ \mathbf{until}\ cnd\ \mathbf{then}\ n') \in vs}{(n,\ vs,\ [\,],\ O)\ \texttt{waiting}}(\texttt{dountil}_{\texttt{waiting}})$$

$$\frac{\mathbf{V}(n,\ \mathbf{if}\ cnd\ \mathbf{then}\ n'\ \mathbf{else}\ n'') \in vs}{(n,\ vs,\ [\,],\ O)\ \texttt{waiting}}(\texttt{ifthen}_{\texttt{waiting}})$$

Figure 9: Dynamics: waiting judgment.

$$\frac{\mathbf{V}(n,\ \mathbf{do}\ a\ \mathbf{then}\ n') \in vs}{(n,\ vs,\ I,\ O) \longmapsto (n',\ vs,\ I,\ a :: O)}(\texttt{doonce}_{\texttt{steps}})$$

$$\frac{\mathbf{V}(n,\ \mathbf{do}\ a\ \mathbf{until}\ cnd\ \mathbf{then}\ n') \in vs}{(n,\ vs,\ true :: I,\ O) \longmapsto (n',\ vs,\ I,\ a :: O)}(\texttt{dountil}^{1}_{\texttt{steps}})$$

$$\frac{\mathbf{V}(n,\ \mathbf{do}\ a\ \mathbf{until}\ cnd\ \mathbf{then}\ n') \in vs}{(n,\ vs,\ false :: I,\ O) \longmapsto (n,\ vs,\ I,\ a :: O)}(\texttt{dountil}^{2}_{\texttt{steps}})$$

$$\frac{\mathbf{V}(n,\ \mathbf{if}\ cnd\ \mathbf{then}\ n'\ \mathbf{else}\ n'') \in vs}{(n,\ vs,\ true :: I,\ O) \longmapsto (n',\ vs,\ I,\ O)}(\texttt{ifthen}^{1}_{\texttt{steps}})$$

$$\frac{\mathbf{V}(n,\ \mathbf{if}\ cnd\ \mathbf{then}\ n'\ \mathbf{else}\ n'') \in vs}{(n,\ vs,\ false :: I,\ O) \longmapsto (n'',\ vs,\ I,\ O)}(\texttt{ifthen}^{2}_{\texttt{steps}})$$

$$\frac{\mathbf{V}(n,\ \mathbf{goto}\ n') \in vs}{(n,\ vs,\ I,\ O) \longmapsto (n',\ vs,\ I,\ O)}(\texttt{goto}_{\texttt{steps}})$$

Figure 10: Dynamics: steps judgment.

Ideally, whenever an instruction graph program is executing, one of two things is true: either the program has ended i.e. n points to a vertex that has **End** as its content, or the program can continue executing i.e. there's a new configuration that the program can proceed to. However, there is actually a third possibility. Since the input to the program was specified as a list, the formal execution of a program could accidentally "run out of inputs" if the list of inputs is empty. In practice, this should never happen unless the robot is malfunctioning, as there is no reason why the robot cannot check a condition to receive input. Because we are forced to deal with this possibility in our model, we call this state "waiting" because it is as if the robot is waiting for input from its sensors.

We describe a configuration where the program has ended "terminated", and we say that when a program can continue to execute that its configuration "steps".

These three judgments are defined precisely in Figures 8-10. They aren't particularly tricky, but more details on the dynamics are available in the appendix.

6. SAFETY

In order to be convinced of a language's correctness it is useful to complete a proof of type safety. Type safety usually deals with progress, which states that an expression in a language is either a final value or can continue to step, and preservation, which states that as a well-typed expression steps, it remains well-typed.

$$\frac{\mathbf{P}(v,\ vs)\ \texttt{valid} \qquad \mathbf{V}(n,\ c) \in v :: vs}{(n,\ v :: vs,\ I,\ O)\ \texttt{cfgvalid}}(\texttt{cfg}_{\texttt{cfgvalid}})$$

Figure 11: Safety: cfgvalid judgment.

Since the ideas of an expression and well-typedness do not apply to the instruction graph language, proving type safety for the instruction graph language is not straightforward.

To do so, we must state a variant of progress and preservation that is more applicable to the ideas of the instruction graph language.

First, though, we should define what a valid configuration is. This is formally defined in Figure 11. Informally, a valid configuration is one where the vertices describe a valid program, and the current vertex is one that exists in the program.

SAFETY 1. *Progress.*
If cfg `cfgvalid`*, then either*

1. *cfg* `terminated`
2. *cfg* `waiting`
3. $\exists\ cfg'\ \texttt{s.t.} cfg \longmapsto cfg'$

SAFETY 2. *Preservation.*
If cfg `cfgvalid` *and* $cfg \longmapsto cfg'$ *then* cfg' `cfgvalid`.

The formulation of progress comes naturally from the discussion of the dynamics. A program midway in execution either needs to keep going, be finished, or perhaps be "waiting" due to missing input.

Preservation is quite natural too. If a configuration is valid, stepping it should neither make the program no longer valid nor step to a vertex that is not part of the program.

Proving progress and preservation for most languages is long and strenuous. As such, the proof is omitted here, but the appendix contains the full entirety of the proof. Instead, we may discuss a basic proof sketch.

The proof for progress is actually not too bad. Given that a configuration is cfgvalid, the vertex corresponding to n must be part of the program. Next, case on the structure on this vertex i.e. what its content is, and whether I is empty or not. As it turns out, every combination of content and whether I is empty or not directly corresponds to a dynamics rule, so the configuration is definitely terminated, waiting, or steps to another configuration.

The proof for preservation is more nontrivial. The proof relies on five separate lemmas, each of which are proved separately. The basic idea is to first establish that n is connected to some nonempty set of vertices, and from that show that n' (the vertex n necessarily connects to) is also connected to some possibly empty set of vertices, which implies that there is a vertex with that index and thus the configuration is still cfgvalid after stepping. As usual, the appendix has a more detailed proof of progress and preservation.

7. INTERPRETER

Finally, in order to show that the instruction graph language can work in practice, we wrote an interpreter that executes instruction graph programs. The interpreter comprises a lexer and parser, as defined by the grammar, a statics checker as defined by the statics, and an interpreter that uses the dynamics to execute the program. In order to simulate the effects of the actions a robot would make as the program executes, the interpreter hooks into the Turtlebot simulator, an open source robotics platform running on ROS. The code for the interpreter, as well as a video demo, can be found at https://github.com/anbenson/instructiongraphs.

8. DISCUSSION

8.1 Surprises

It is interesting to note how easily a data structure like a graph can be molded into the confines of a programming language. Although the grammar of the language is unlike most others, the formulation and implementation of the statics are almost exactly the same as a breadth-first traversal of the graph to determine connectiveness. There is a surprising relationship between inference rules governing invariants and algorithms that take advantage of invariants.

8.2 Conclusions

We have proposed a formalization of the instruction graph data structure as a domain specific programming language, the *instruction graph language*. Following that, we have proven theorems of type safety for the language. One consequence of the fact that the instruction graph language is safe is that because the instruction graph data structure, as evaluated internally by robots in [1], and the instruction graph language are equivalent, as they both express the same thing, we can extend the theorems of type safety to the instruction graph data structure. This means that the instruction graph data structure is safe - execution of it will not get "stuck" or transition into an invalid state not predicted by the statics.

We have also created an interpreter to show feasibility of the instruction graph language.

8.3 Future Work

While instruction graphs are expressive enough to describe many robotic tasks, they are not a particularly pleasant one. There is definitely room for improvement - for example, the instruction graph language lacks any means of abstraction. In other words, there is no way to reuse "subgraphs" of the instruction graph in other parts of the instruction graph without copying over the entire subgraph. Fortunately, further work in instruction graphs should be made easier by a formalization of instruction graphs, and perhaps by a concrete interpreter to execute them.

9. ACKNOWLEDGMENTS

The author would like to thank his advisor, Jonathan Aldrich, for all the advice and help given throughout this semester-long project. The author would also like to thank Joshua Sunshine and Cyrus Omar, who also contributed useful feedback and guidance about this project.

10. REFERENCES

[1] C. Mericli, S. Klee, J. Paparian, and M. Veloso. An Interactive Approach for Situated Task Specification through Verbal Instructions. In *Proceedings of AAMAS'14, the Thirteenth International Joint Conference on Autonomous Agents and Multi-Agent Systems*, Paris, France, May 2014.

Appendix A: Instruction Graph Grammar

`Program`	p	$::=$	$\mathbf{P}(v,\ vs)$	programs
`Vertices`	vs	$::=$	nil	empty
			v **::** vs	cons
`Vertex`	v	$::=$	$\mathbf{V}(n,\ c)$	vertex
`Content`	c	$::=$	**do** a **then** n	single action
			do a **until** cnd **then** n	open loop action
			if cnd **then** n **else** n	conditional
			goto n	goto
			end	termination

We let $n \in \mathbb{Z}$, the integers.

We let $a \in$ `Action`, a sort describing classes of actions, like movement, that a robot might be able to perform. A grammar defining `Action` is assumed.

We let $cnd \in$ `Condition`, a sort describing classes of conditions, like whether an object is some distance ahead, that a robot might be able to detect. A grammar defining `Condition` is assumed.

Appendix B: Instruction Graph Statics

1 Validity

p `valid` means that the Program p is a valid program.

$$\frac{(\mathbf{V}(s,\ c) :: vs,\ U)\ \texttt{defined} \qquad (\mathbf{V}(s,\ c) :: vs,\ \emptyset,\ s,\ U)\ \texttt{connected}}{\mathbf{P}(\mathbf{V}(s,\ c),\ vs)\ \texttt{valid}}(\mathrm{P}_{\texttt{valid}})$$

2 Defined

We let $U \subseteq \mathbb{Z}$ be a subset of the integers.

$(vs,\ U)$ `defined` means that the Vertices vs define exactly the set U of vertex indices.

$$\frac{}{(nil,\ \{\ \})\ \texttt{defined}}(\texttt{nil}_{\texttt{defined}})$$

$$\frac{(vs,\ U)\ \texttt{defined} \qquad n \notin U}{(\mathbf{V}(n,\ c) :: vs,\ U \cup \{n\})\ \texttt{defined}}(\texttt{cons}_{\texttt{defined}})$$

3 Connected

We let $U \subseteq \mathbb{Z}$ be a subset of the integers.

$(vs,\ U_v,\ n,\ U)$ `connected` means that there exists a path from the vertex represented by n to each vertex represented by an index in U without going through any vertex in U_v. By "represented" we mean that vs contains a vertex for that index.

$$\frac{(vs,\ U)\ \texttt{defined} \qquad U_v \subseteq U \qquad n \in U_v}{(vs,\ U_v,\ n,\ \emptyset)\ \texttt{connected}}(\texttt{visited}_{\texttt{connected}})$$

$$\frac{(vs,\ U)\ \texttt{defined} \qquad U_v \subseteq U \qquad \mathbf{V}(n,\ \mathbf{end}) \in vs \qquad n \notin U_v}{(vs,\ U_v,\ n,\ \{n\})\ \texttt{connected}}(\texttt{end}_{\texttt{connected}})$$

$$\frac{\begin{array}{c}\mathbf{V}(n,\ \mathbf{do}\ a\ \mathbf{then}\ n') \in vs \\ (vs,\ U_v \cup \{n\},\ n',\ U)\ \texttt{connected} \qquad n \notin U_v\end{array}}{(vs,\ U_v,\ n,\ U \cup \{n\})\ \texttt{connected}}(\texttt{doonce}_{\texttt{connected}})$$

$$\frac{\begin{array}{c}\mathbf{V}(n,\ \mathbf{do}\ a\ \mathbf{until}\ cnd\ \mathbf{then}\ n') \in vs \\ (vs,\ U_v \cup \{n\},\ n',\ U)\ \texttt{connected} \qquad n \notin U_v\end{array}}{(vs,\ U_v,\ n,\ U \cup \{n\})\ \texttt{connected}}(\texttt{dountil}_{\texttt{connected}})$$

$$\frac{\begin{array}{c}\mathbf{V}(n,\ \mathbf{if}\ cnd\ \mathbf{then}\ n'\ \mathbf{else}\ n'') \in vs \qquad (vs,\ U_v \cup \{n\},\ n',\ U)\ \texttt{connected} \\ (vs,\ U_v \cup U \cup \{n\},\ n'',\ U')\ \texttt{connected} \qquad n \notin U_v\end{array}}{(vs,\ U_v,\ n,\ U \cup U' \cup \{n\})\ \texttt{connected}}(\texttt{ifthen}_{\texttt{connected}})$$

$$\frac{\mathbf{V}(n,\ \mathbf{goto}\ n') \in vs \qquad (vs,\ U_v \cup \{n\},\ n',\ U)\ \texttt{connected} \qquad n \notin U_v}{(vs,\ U_v,\ n,\ U \cup \{n\})\ \texttt{connected}}(\texttt{goto}_{\texttt{connected}})$$

Appendix C: Instruction Graph Dynamics

1 Configuration

To describe a state midway through execution, we define

$$\texttt{Configuration} \quad \mathit{cfg} \quad ::= \quad (n,\ \mathit{vs},\ I,\ O) \qquad \text{configurations}$$

where $n \in \mathbb{Z}$, the integers, $\mathit{vs} \in \texttt{Vertices}$, I is a *bool* list, representing the input used to satisfy a `Condition` cnd, and O is an `Action` list, representing the ordered (but reversed) list of actions that are executed.

2 Terminated

$(n,\ \mathit{vs},\ I,\ O)$ `terminated` means that the state with vertex represented by n in vertices vs with remaining input I and current output O is in a finished state for the program execution context.

$$\frac{\mathbf{V}(n,\ \mathbf{end}) \in \mathit{vs}}{(n,\ \mathit{vs},\ I,\ O)\ \texttt{terminated}}(\texttt{cfg}_{\texttt{terminated}})$$

3 Waiting

$(n,\ \mathit{vs},\ I,\ O)$ `waiting` means that the state with vertex represented by n in vertices vs with remaining input I and current output O cannot proceed, as it requires more input to continue.

$$\frac{\mathbf{V}(n,\ \mathbf{do}\ a\ \mathbf{until}\ cnd\ \mathbf{then}\ n') \in vs}{(n,\ vs,\ [\,],\ O)\ \texttt{waiting}}(\texttt{dountil}_{\texttt{waiting}})$$

$$\frac{\mathbf{V}(n,\ \mathbf{if}\ cnd\ \mathbf{then}\ n'\ \mathbf{else}\ n'') \in vs}{(n,\ vs,\ [\,],\ O)\ \texttt{waiting}}(\texttt{ifthen}_{\texttt{waiting}})$$

4 Steps

$(n,\ vs,\ I,\ O) \longmapsto (n',\ vs,\ I',\ O')$ means that the state with vertex represented by n in vertices vs with remaining input I and current output O continues to the state with vertex represented by n' in vertices vs with remaining input I' and current output O'.

$$\frac{\mathbf{V}(n,\ \mathbf{do}\ a\ \mathbf{then}\ n') \in vs}{(n,\ vs,\ I,\ O) \longmapsto (n',\ vs,\ I,\ a :: O)}(\texttt{doonce}_{\texttt{steps}})$$

$$\frac{\mathbf{V}(n,\ \mathbf{do}\ a\ \mathbf{until}\ cnd\ \mathbf{then}\ n') \in vs}{(n,\ vs,\ true :: I,\ O) \longmapsto (n',\ vs,\ I,\ a :: O)}(\texttt{dountil}^{1}_{\texttt{steps}})$$

$$\frac{\mathbf{V}(n,\ \mathbf{do}\ a\ \mathbf{until}\ cnd\ \mathbf{then}\ n') \in vs}{(n,\ vs,\ false :: I,\ O) \longmapsto (n,\ vs,\ I,\ a :: O)}(\texttt{dountil}^{2}_{\texttt{steps}})$$

$$\frac{\mathbf{V}(n,\ \mathbf{if}\ cnd\ \mathbf{then}\ n'\ \mathbf{else}\ n'') \in vs}{(n,\ vs,\ true :: I,\ O) \longmapsto (n',\ vs,\ I,\ O)}(\texttt{ifthen}^{1}_{\texttt{steps}})$$

$$\frac{\mathbf{V}(n,\ \mathbf{if}\ cnd\ \mathbf{then}\ n'\ \mathbf{else}\ n'') \in vs}{(n,\ vs,\ false :: I,\ O) \longmapsto (n'',\ vs,\ I,\ O)}(\texttt{ifthen}^{2}_{\texttt{steps}})$$

$$\frac{\mathbf{V}(n,\ \mathbf{goto}\ n') \in vs}{(n,\ vs,\ I,\ O) \longmapsto (n',\ vs,\ I,\ O)}(\texttt{goto}_{\texttt{steps}})$$

Appendix D: Instruction Graph Proofs

1 cfgvalid

cfg `cfgvalid` means that the configuration *cfg* is a valid configuration.

$$\frac{\mathbf{P}(v,\ vs)\ \texttt{valid} \qquad \mathbf{V}(n,\ c) \in v :: vs}{(n,\ v :: vs,\ I,\ O)\ \texttt{cfgvalid}}(\texttt{cfg}_{\texttt{cfgvalid}})$$

2 Progress

Theorem 1. *If cfg* `cfgvalid`*, then either*

1. *cfg* `terminated`
2. *cfg* `waiting`
3. $\exists\ cfg'\ \texttt{s.t.} cfg \longmapsto cfg'$

2.1 Proof of Progress

We proceed by case analysis on the judgment *cfg* `cfgvalid`. There is only one rule that concludes *cfg* `cfgvalid`:

$$\frac{\mathbf{P}(v,\ vs)\ \texttt{valid} \qquad \mathbf{V}(n,\ c) \in v :: vs}{(n,\ v :: vs,\ I,\ O)\ \texttt{cfgvalid}}$$

So we know *cfg* is of the form $(n,\ v :: vs,\ I,\ O)$ and $\mathbf{V}(n,\ c) \in v :: vs$. We continue by structural induction on c, which is of the sort `Content`.

Case c is $\textbf{do}\ a\ \textbf{then}\ n'$:

Then by the rule

$$\frac{\mathbf{V}(n,\ \textbf{do}\ a\ \textbf{then}\ n') \in vs}{(n,\ vs,\ I,\ O) \longmapsto (n',\ vs,\ I,\ a :: O)}$$

we can conclude $(n,\ v :: vs,\ I,\ O) \longmapsto (n',\ v :: vs,\ I,\ a :: O)$.

Case c is $\textbf{do}\ a\ \textbf{until}\ cnd\ \textbf{then}\ n'$:

We use structural induction on I.

Subcase I is $[\,]$:

Then by the rule

$$\frac{\mathbf{V}(n,\ \textbf{do}\ a\ \textbf{until}\ cnd\ \textbf{then}\ n') \in vs}{(n,\ vs,\ [\,],\ O)\ \texttt{waiting}}$$

we can conclude $(n,\ v :: vs,\ I,\ O)\ \texttt{waiting}$.

Subcase I is $true :: I'$:

Then by the rule

$$\frac{\mathbf{V}(n,\ \textbf{do}\ a\ \textbf{until}\ cnd\ \textbf{then}\ n') \in vs}{(n,\ vs,\ true :: I,\ O) \longmapsto (n',\ vs,\ I,\ a :: O)}$$

we can conclude $(n,\ v :: vs,\ I,\ O) \longmapsto (n',\ v :: vs,\ I',\ a :: O)$.

Subcase I is $false :: I'$:

Then by the rule

$$\frac{\mathbf{V}(n,\ \mathbf{do}\ a\ \mathbf{until}\ \mathit{cnd}\ \mathbf{then}\ n') \in \mathit{vs}}{(n,\ \mathit{vs},\ \mathit{false} :: I,\ O) \longmapsto (n,\ \mathit{vs},\ I,\ a :: O)}$$

we can conclude $(n,\ v :: \mathit{vs},\ I,\ O) \longmapsto (n,\ v :: \mathit{vs},\ I',\ a :: O)$.

Case c is **if** cnd **then** n' **else** n'':

We use structural induction on I.

Subcase I is $[\,]$:

Then by the rule

$$\frac{\mathbf{V}(n,\ \mathbf{if}\ \mathit{cnd}\ \mathbf{then}\ n'\ \mathbf{else}\ n'') \in \mathit{vs}}{(n,\ \mathit{vs},\ [\,],\ O)\ \texttt{waiting}}$$

we can conclude $(n,\ v :: \mathit{vs},\ I,\ O)\ \texttt{waiting}$.

Subcase I is $\mathit{true} :: I'$:

Then by the rule

$$\frac{\mathbf{V}(n,\ \mathbf{if}\ \mathit{cnd}\ \mathbf{then}\ n'\ \mathbf{else}\ n'') \in \mathit{vs}}{(n,\ \mathit{vs},\ \mathit{true} :: I,\ O) \longmapsto (n',\ \mathit{vs},\ I,\ O)}$$

we can conclude $(n,\ v :: \mathit{vs},\ I,\ O) \longmapsto (n',\ v :: \mathit{vs},\ I',\ O)$.

Subcase I is $\mathit{false} :: I'$:

Then by the rule

$$\frac{\mathbf{V}(n,\ \mathbf{if}\ \mathit{cnd}\ \mathbf{then}\ n'\ \mathbf{else}\ n'') \in \mathit{vs}}{(n,\ \mathit{vs},\ \mathit{false} :: I,\ O) \longmapsto (n'',\ \mathit{vs},\ I,\ O)}$$

we can conclude $(n,\ v :: vs,\ I,\ O) \longmapsto (n'',\ v :: vs,\ I',\ O)$.

Case c is **goto** n':

Then by the rule

$$\frac{\mathbf{V}(n,\ \mathbf{goto}\ n') \in vs}{(n,\ vs,\ I,\ O) \longmapsto (n',\ vs,\ I,\ O)}$$

we can conclude $(n,\ v :: vs,\ I,\ O) \longmapsto (n',\ v :: vs,\ I,\ O)$.

Case c is **end**:

Then by the rule

$$\frac{\mathbf{V}(n,\ \mathbf{end}) \in vs}{(n,\ vs,\ I,\ O)\ \texttt{terminated}}$$

we can conclude $(n,\ v :: vs,\ I,\ O)$ `terminated`.

3 Preservation

Theorem 2. *If cfg* `cfgvalid` *and cfg* $\longmapsto$ *cfg′ then cfg′* `cfgvalid`*.*

3.1 Lemma 1

If $(vs,\ U_v,\ n,\ U)$ `connected` then $\forall n' \in U\ .\ \exists\ U'_v,\ U'$ such that U' is nonempty and $(vs,\ U'_v,\ n',\ U')$ `connected`.

Proof: We proceed by rule induction on $(vs,\ U_v,\ n,\ U)$ `connected`.

Case $\dfrac{(vs,\ U)\ \texttt{defined} \qquad U_v \subseteq U \qquad n \in U_v}{(vs,\ U_v,\ n,\ \emptyset)\ \texttt{connected}}$

Then U is the empty set so the lemma is vacuously true.

$$\text{Case}\ \frac{(vs,\ U)\ \texttt{defined} \qquad U_v \subseteq U \qquad \mathbf{V(}n,\ \mathbf{end)} \in vs \qquad n \notin U_v}{(vs,\ U_v,\ n,\ \{n\})\ \texttt{connected}}$$

Then U contains exactly n. But we already have $(vs,\ U_v,\ n,\ \{n\})$ `connected` so the lemma is satisfied.

$$\text{Case}\ \frac{\begin{array}{c}\mathbf{V(}n,\ \mathbf{do}\ a\ \mathbf{then}\ n'\mathbf{)} \in vs \\ (vs,\ U_v \cup \{n\},\ n',\ U)\ \texttt{connected} \qquad n \notin U_v\end{array}}{(vs,\ U_v,\ n,\ U \cup \{n\})\ \texttt{connected}}$$

Then by the inductive hypothesis, since $(vs,\ U_v \cup \{n\},\ n',\ U)$ `connected` we know the lemma is satisfied for all $n' \in U$. All that's left is to show it is satisfied for n, but we have $(vs,\ U_v,\ n,\ U \cup \{n\})$ `connected`.

$$\text{Case}\ \frac{\begin{array}{c}\mathbf{V(}n,\ \mathbf{do}\ a\ \mathbf{until}\ cnd\ \mathbf{then}\ n'\mathbf{)} \in vs \\ (vs,\ U_v \cup \{n\},\ n',\ U)\ \texttt{connected} \qquad n \notin U_v\end{array}}{(vs,\ U_v,\ n,\ U \cup \{n\})\ \texttt{connected}}$$

Analogous to the case above.

$$\text{Case}\ \frac{\mathbf{V(}n,\ \mathbf{goto}\ n'\mathbf{)} \in vs \qquad (vs,\ U_v \cup \{n\},\ n',\ U)\ \texttt{connected} \qquad n \notin U_v}{(vs,\ U_v,\ n,\ U \cup \{n\})\ \texttt{connected}}$$

Analogous to the case above.

$$\text{Case}\ \frac{\begin{array}{c}\mathbf{V(}n,\ \mathbf{if}\ cnd\ \mathbf{then}\ n'\ \mathbf{else}\ n''\mathbf{)} \in vs \qquad (vs,\ U_v \cup \{n\},\ n',\ U)\ \texttt{connected} \\ (vs,\ U_v \cup U \cup \{n\},\ n'',\ U')\ \texttt{connected} \qquad n \notin U_v\end{array}}{(vs,\ U_v,\ n,\ U \cup U' \cup \{n\})\ \texttt{connected}}$$

Then by the inductive hypothesis, since $(vs,\ U_v \cup \{n\},\ n',\ U)$ `connected` and $(vs,\ U_v \cup U \cup \{n\},\ n'',\ U')$ `connected` we know the lemma is satisfied for all $n' \in U \cup U'$. All that's left is to show it is satisfied for n, but we have $(vs,\ U_v,\ n,\ U \cup U' \cup \{n\})$ `connected`.

3.2 Lemma 2

If $\mathbf{V}(n,\ c) \in vs$ and $(vs,\ U)$ `defined` then $n \in U$.

Proof: We proceed by rule induction on $(vs,\ U)$ `defined`.

Case $\dfrac{}{(nil,\ \{\ \})\ \texttt{defined}}$

But vs is nil, so the lemma is vacuously true.

Case $\dfrac{(vs,\ U)\ \texttt{defined} \qquad n' \notin U}{(\mathbf{V}(n',\ c) :: vs,\ U \cup \{n'\})\ \texttt{defined}}$

(n is replaced by n' in this statement of the rule to avoid ambiguity.)

The vertex in question, $\mathbf{V}(n,\ c)$, is either $\mathbf{V}(n',\ c)$ or $\in vs$. If it is the former, we are done since $n' \in U \cup \{n'\}$. If it is the latter, then by the inductive hypothesis, since $(vs,\ U)$ `defined` we know $n \in U$ so $n \in U \cup \{n'\}$.

3.3 Lemma 3

If $(vs,\ U)$ `defined` and $n \in U$ then $\mathbf{V}(n,\ c) \in vs$ for some c.

Proof: We proceed by rule induction on $(vs,\ U)$ `defined`.

Case $\dfrac{}{(nil,\ \{\ \})\ \texttt{defined}}$

But U is empty so the lemma is vacuously true.

Case $\dfrac{(vs,\ U)\ \texttt{defined} \qquad n' \notin U}{(\mathbf{V}(n',\ c) :: vs,\ U \cup \{n'\})\ \texttt{defined}}$

(n is replaced by n' in this statement of the rule to avoid ambiguity.)

n is either $\in U$ or is n'. We know it's not both since $n' \notin U$. If it's the former, then by the inductive hypothesis, since $(vs,\ U)$ `defined`, we know there is a $\mathbf{V}(n,\ c) \in vs$, which is also in $\mathbf{V}(n',\ c) :: vs$. If it's the latter, then clearly $\mathbf{V}(n',\ c) \in \mathbf{V}(n',\ c) :: vs$.

3.4 Lemma 4

If $(vs,\ U)$ `defined` and $\mathbf{V}(n,\ c) \in vs$ and $\mathbf{V}(n,\ c') \in vs$ then $c = c'$.

Proof: We proceed by induction on $(vs,\ U)$ `defined`.

Case $\dfrac{}{(nil,\ \{\ \})\ \texttt{defined}}$

But vs is nil so the lemma is vacuously true.

Case $\dfrac{(vs,\ U)\ \texttt{defined} \qquad n' \notin U}{(\mathbf{V}(n',\ c) :: vs,\ U \cup \{n'\})\ \texttt{defined}}$

(n is replaced by n' in this statement of the rule to avoid ambiguity.)

Suppose that $\mathbf{V}(n,\ c)$ and $\mathbf{V}(n,\ c')$ are both $\mathbf{V}(n',\ c)$. Then clearly $c = c'$.

Suppose instead that one of the two is $\mathbf{V}(n',\ c)$ (so $n = n'$) and the other is $\in vs$. Then by Lemma 2, $n \in U$. But $n' \notin U$, contradiction.

Suppose lastly that both are $\in vs$. Then by the inductive hypothesis, $c = c'$.

3.5 Lemma 5

If $(vs,\ U_v,\ n,\ U)$ `connected`, then $\forall\ n' \in U_v\ .\ \exists\ c'$ such that $\mathbf{V}(n',\ c') \in vs$.

Proof: We proceed by rule induction on $(vs,\ U_v,\ n,\ U)$ `connected`.

$$\text{Case}\ \frac{(vs,\ U)\ \texttt{defined} \qquad U_v \subseteq U \qquad n \in U_v}{(vs,\ U_v,\ n,\ \emptyset)\ \texttt{connected}}$$

Since $U_v \subseteq U$, it suffices to check this is true for every element in U. Since we have $(vs,\ U)$ `defined`, we have this property by Lemma 3.

$$\text{Case}\ \frac{(vs,\ U)\ \texttt{defined} \qquad U_v \subseteq U \qquad \mathbf{V}(n,\ \mathbf{end}) \in vs \qquad n \notin U_v}{(vs,\ U_v,\ n,\ \{n\})\ \texttt{connected}}$$

Analogous to the case above.

$$\text{Case}\ \frac{\begin{array}{c}\mathbf{V}(n,\ \mathbf{do}\ a\ \mathbf{then}\ n') \in vs \\ (vs,\ U_v \cup \{n\},\ n',\ U)\ \texttt{connected} \qquad n \notin U_v\end{array}}{(vs,\ U_v,\ n,\ U \cup \{n\})\ \texttt{connected}}$$

Then by the inductive hypothesis, since $(vs,\ U_v \cup \{n\},\ n',\ U)$ `connected` we know the lemma is satisfied for all $n' \in U_v \cup \{n\}$. Since $U_v \subseteq U_v \cup \{n\}$, this is also true for all $n' \in U_v$.

$$\text{Case}\ \frac{\begin{array}{c}\mathbf{V}(n,\ \mathbf{do}\ a\ \mathbf{until}\ cnd\ \mathbf{then}\ n') \in vs \\ (vs,\ U_v \cup \{n\},\ n',\ U)\ \texttt{connected} \qquad n \notin U_v\end{array}}{(vs,\ U_v,\ n,\ U \cup \{n\})\ \texttt{connected}}$$

Analogous to the case above.

$$\text{Case}\ \frac{\mathbf{V}(n,\ \mathbf{goto}\ n') \in vs \qquad (vs,\ U_v \cup \{n\},\ n',\ U)\ \texttt{connected} \qquad n \notin U_v}{(vs,\ U_v,\ n,\ U \cup \{n\})\ \texttt{connected}}$$

Analogous to the case above.

$$\text{Case}\ \frac{\mathbf{V}(n,\ \textbf{if}\ cnd\ \textbf{then}\ n'\ \textbf{else}\ n'') \in vs \qquad (vs,\ U_v \cup \{n\},\ n',\ U)\ \texttt{connected} \qquad (vs,\ U_v \cup U \cup \{n\},\ n'',\ U')\ \texttt{connected} \qquad n \notin U_v}{(vs,\ U_v,\ n,\ U \cup U' \cup \{n\})\ \texttt{connected}}$$

Analogous to the case above.

3.6 Proof of Preservation

Recall that we stated preservation as "If *cfg* `cfgvalid` and $cfg \longmapsto cfg'$ then cfg' `cfgvalid`."

We begin by case analyzing on *cfg* `cfgvalid` in order to conclude some important facts.

The only case is

$$\frac{\mathbf{P}(v,\ vs)\ \texttt{valid} \qquad \mathbf{V}(n,\ c) \in v :: vs}{(n,\ v :: vs,\ I,\ O)\ \texttt{cfgvalid}}$$

so we can conclude:

$$cfg = (n,\ v :: vs,\ I,\ O) \tag{1}$$

$$\mathbf{P}(v,\ vs)\ \texttt{valid} \tag{2}$$

$$\mathbf{V}(n,\ c) \in v :: vs \tag{3}$$

We case analyze on $\mathbf{P}(v,\ vs)$ `valid`. The only case is

$$\frac{(\mathbf{V}(s,\ c_s) :: vs,\ U)\ \texttt{defined} \qquad (\mathbf{V}(s,\ c_s) :: vs,\ \emptyset,\ s,\ U)\ \texttt{connected}}{\mathbf{P}(\mathbf{V}(s,\ c_s),\ vs)\ \texttt{valid}}$$

so we can conclude:

$$v = \mathbf{V}(s,\ c_s) \tag{4}$$

$$(\mathbf{V}(s,\ c_s) :: vs,\ U)\ \texttt{defined} \tag{5}$$

$$(\mathbf{V}(s,\ c_s) :: vs,\ \emptyset,\ s,\ U)\ \texttt{connected} \tag{6}$$

From (3), (4), and (5), using Lemma 2 we can conclude

$$n \in U \tag{7}$$

From (6) and (7), using Lemma 1 we can conclude $\exists\ U'_v,\ U'$ where U' is not empty such that

$$(v :: vs,\ U'_v,\ n,\ U')\ \texttt{connected} \tag{8}$$

We continue by case analyzing on $cfg \longmapsto cfg'$.

Case $\dfrac{\mathbf{V}(n,\ \mathbf{do}\ a\ \mathbf{then}\ n') \in vs'}{(n,\ vs',\ I,\ O) \longmapsto (n',\ vs',\ I,\ a :: O)}$

So taking into account (1), cfg is of the form $(n,\ v :: vs,\ I,\ O)$ and cfg' is of the form $(n',\ v :: vs,\ I,\ a :: O)$. We do another case analysis on (8).

Subcase $\dfrac{(v :: vs,\ U)\ \texttt{defined} \qquad U'_v \subseteq U \qquad n \in U'_v}{(v :: vs,\ U'_v,\ n,\ \emptyset)\ \texttt{connected}}$

But U' is not empty, contradiction.

All but one of all of the other subcases have a premise of the form $\mathbf{V}(n,\ c') \in v :: vs$ where c' is not **do** a **then** n'. This is a contradiction by Lemma 4 since we have $\mathbf{V}(n,\ \textbf{do}\ a\ \textbf{then}\ n') \in v :: vs$.

Thus the only subcase left is

$$\frac{\begin{array}{c}\mathbf{V}(n,\ \textbf{do}\ a\ \textbf{then}\ n') \in vs \\ (v :: vs,\ U'_v \cup \{n\},\ n',\ U'')\ \texttt{connected} \qquad n \notin U'_v\end{array}}{(v :: vs,\ U'_v,\ n,\ U'' \cup \{n\})\ \texttt{connected}}$$

If we let U''_v be $U'_v \cup \{n\}$, then we can case analyze on $(v :: vs,\ U''_v,\ n',\ U'')\ \texttt{connected}$:

Sub-subcase $\dfrac{(v :: vs,\ U)\ \texttt{defined} \qquad U''_v \subseteq U \qquad n' \in U''_v}{(v :: vs,\ U''_v,\ n',\ \emptyset)\ \texttt{connected}}$

Then since we have $n' \in U''_v$, by Lemma 5, $\exists\ c'$ such that $\mathbf{V}(n',\ c') \in v :: vs$.

In any of the other sub-subcases, one of the premises gives us $\mathbf{V}(n',\ c') \in v :: vs$ for some c'.

So regardless of the sub-subcase, we have $\mathbf{V}(n',\ c') \in v :: vs$. Combining with (2), we can conclude $(n',\ v :: vs,\ I,\ a :: O)\ \texttt{cfgvalid}$ as desired.

Case $\dfrac{\mathbf{V}(n,\ \textbf{do}\ a\ \textbf{until}\ cnd\ \textbf{then}\ n') \in vs}{(n,\ vs,\ true :: I,\ O) \longmapsto (n',\ vs,\ I,\ a :: O)}$

So taking into account (1), *cfg* is of the form $(n,\ v :: vs,\ true :: I',\ O)$ and *cfg'* is of the form $(n',\ v :: vs,\ I',\ a :: O)$. The rest of the proof for this case is analogous to above except c is **do** a **until** cnd **then** n' and the end conclusion is that $(n',\ v :: vs,\ I',\ a :: O)\ \texttt{cfgvalid}$.

Case $\dfrac{\mathbf{V}(n,\ \textbf{do}\ a\ \textbf{until}\ cnd\ \textbf{then}\ n') \in vs}{(n,\ vs,\ false :: I,\ O) \longmapsto (n,\ vs,\ I,\ a :: O)}$

So taking into account (1), *cfg* is of the form $(n,\ v :: vs,\ \mathit{false} :: I',\ O)$ and *cfg*$'$ is of the form $(n,\ v :: vs,\ I',\ a :: O)$. The rest of the proof for this case is analogous to above except c is **do** a **until** *cnd* **then** n' and the end conclusion is that $(n,\ v :: vs,\ I',\ a :: O)$ `cfgvalid`.

$$\text{Case}\ \frac{\mathbf{V}(n,\ \mathbf{goto}\ n') \in vs}{(n,\ vs,\ I,\ O) \longmapsto (n',\ vs,\ I,\ O)}$$

So taking into account (1), *cfg* is of the form $(n,\ v :: vs,\ I,\ O)$ and *cfg*$'$ is of the form $(n',\ v :: vs,\ I,\ O)$. The rest of the proof for this case is analogous to above except c is **goto** n' and the end conclusion is that $(n',\ v :: vs,\ I,\ O)$ `cfgvalid`.

$$\text{Case}\ \frac{\mathbf{V}(n,\ \mathbf{if}\ \mathit{cnd}\ \mathbf{then}\ n'\ \mathbf{else}\ n'') \in vs}{(n,\ vs,\ \mathit{true} :: I,\ O) \longmapsto (n',\ vs,\ I,\ O)}$$

So taking into account (1), *cfg* is of the form $(n,\ v :: vs,\ \mathit{true} :: I',\ O)$ and *cfg*$'$ is of the form $(n',\ v :: vs,\ I',\ O)$. The rest of the proof for this case is analogous to above except c is **if** *cnd* **then** n' **else** n'' and the end conclusion is that $(n',\ v :: vs,\ I',\ a :: O)$ `cfgvalid`.

$$\text{Case}\ \frac{\mathbf{V}(n,\ \mathbf{if}\ \mathit{cnd}\ \mathbf{then}\ n'\ \mathbf{else}\ n'') \in vs}{(n,\ vs,\ \mathit{false} :: I,\ O) \longmapsto (n'',\ vs,\ I,\ O)}$$

So taking into account (1), *cfg* is of the form $(n,\ v :: vs,\ \mathit{false} :: I',\ O)$ and *cfg*$'$ is of the form $(n'',\ v :: vs,\ I',\ O)$. We do another case analysis on (8).

$$\text{Subcase}\ \frac{(v :: vs,\ U)\ \texttt{defined} \qquad U'_v \subseteq U \qquad n \in U'_v}{(v :: vs,\ U'_v,\ n,\ \emptyset)\ \texttt{connected}}$$

But U' is not empty, contradiction.

All but one of all of the other subcases have a premise of the form $\mathbf{V}(n,\ c') \in v :: vs$ where c' is not **if** *cnd* **then** n' **else** n''. This is a contradiction by Lemma 4 since we have $\mathbf{V}(n,\ \textbf{if}\ cnd\ \textbf{then}\ n'\ \textbf{else}\ n'') \in v :: vs$.

Thus the only subcase left is

$$\frac{\begin{array}{c}\mathbf{V}(n,\ \textbf{if}\ cnd\ \textbf{then}\ n'\ \textbf{else}\ n'') \in v :: vs \\ (v :: vs,\ U'_v \cup \{n\},\ n',\ U)\ \texttt{connected} \\ (v :: vs,\ U'_v \cup U \cup \{n\},\ n'',\ U'')\ \texttt{connected} \qquad n \notin U'_v\end{array}}{(v :: vs,\ U'_v,\ n,\ U \cup U'' \cup \{n\})\ \texttt{connected}}$$

If we let U''_v be $U'_v \cup U \cup \{n\}$, then we can case analyze on $(v :: vs,\ U''_v,\ n'',\ U'')$ `connected`:

Sub-subcase $\dfrac{(v :: vs,\ U)\ \texttt{defined} \qquad U''_v \subseteq U \qquad n'' \in U''_v}{(v :: vs,\ U''_v,\ n'',\ \emptyset)\ \texttt{connected}}$

Then since we have $n'' \in U''_v$, by Lemma 5, $\exists\ c'$ such that $\mathbf{V}(n'',\ c') \in v :: vs$.

In any of the other sub-subcases, one of the premises gives us $\mathbf{V}(n'',\ c') \in v :: vs$ for some c'.

So regardless of the sub-subcase, we have $\mathbf{V}(n'',\ c') \in v :: vs$. Combining with (2), we can conclude $(n'',\ v :: vs,\ I',\ O)$ `cfgvalid` as desired.