Verifying a Copying Garbage Collector in GP 2

Gia S. Wulandari*[1,2] and Detlef Plump[1]

[1] Department of Computer Science, University of York, UK
[2] School of Computing, Telkom University, Indonesia

Abstract. Cheney's copying garbage collector is regarded as a challenging test case for formal approaches to the verification of imperative programs with pointers. The algorithm works for possibly cyclic data structures with unrestricted sharing which cannot be handled by standard separation logics. In addition, the algorithm relocates data and requires establishing an isomorphism between the initial and the final data structure of a program run.
We present an implementation of Cheney's garbage collector in the graph programming language GP 2 and a proof that it is totally correct. Our proof is shorter and less complicated than comparable proofs in the literature. This is partly due to the fact that the GP 2 program abstracts from details of memory management such as address arithmetic. We use sound proof rules previously employed in the verification of GP 2 programs but treat assertions semantically because current assertion languages for graph transformation cannot express the existence of an isomorphism between initial and final graphs.

1 Introduction

Poskitt and Plump developed Hoare-style proof systems for verifying the partial and total correctness of graph programs and showed that their proof calculi are sound with respect to the operational semantics of graph programs in the language GP 2 [13, 12]. In these calculi, pre- and postconditions of programs are so-called E-conditions which extend nested graph conditions with support for expressions. E-conditions are limited to specify first-order graph properties and cannot express non-local properties such as connectedness or the existence of arbitrary-length paths. M-conditions [15] overcome this limitation in that they express monadic second-order properties of graphs.

In this paper, we present the verification of a graph program that cannot be proved correct by using E- or M-conditions because its correctness requires to establish a certain isomorphism between input and output graphs. We implement Cheney's copying garbage collector [3] in the graph programming language GP 2 and prove that it is totally correct. Cheney's algorithm is regarded as a challenge for formal approaches to verifying pointer programs. This is because it works for possibly cyclic data structures with unrestricted sharing which cannot be handled by standard separation logics [16, 5]. In addition, the algorithm relocates

* Supported by Indonesia Endowment Fund for Education (LPDP)

data which requires establishing an isomorphism between the initial and the final data structure of a program run.

While cycles and unrestricted sharing are not a problem for formal assertions based on nested graph conditions, the existence of an isomorphism between initial and final graphs of program runs cannot be expressed with such assertions. Therefore we treat assertions semantically, without a formal language, but use sound proof rules that were previously employed in GP 2 verification.

The remainder of this paper is structured is follows: Section 2 briefly describes the graph programming language GP 2, followed by Section 3, where we introduce the basic notions of graph program verification. In Section 4, we present an implementation of Cheney's copying garbage collector in GP 2. In Section 5, we precisely specify the garbage collector by pre- and postconditions. In Section 6, we prove that our implementation is partially correct, will terminate and cannot fail. In Section 7, we argue that our proof of partial correctness is shorter and less complicated than comparable proofs in the literature. Finally, we conclude and give some topics for future work in Section 8.

2 Graph Programs

We briefly describe a subset of the graph programming language GP 2 that is sufficient for our case study. A full definition of GP 2, including a formal operational semantics, can be found in [11]. The language is implemented by a compiler generating C code [1].

The principal programming constructs in GP 2 are graph-transformation rules labelled with expressions. For example, the program `cheney` in Fig. 2 contains the declarations of five rules. Rules operate on *host graphs* whose nodes and edges are labelled with heterogeneous lists of integers and character strings. Variables in rules are typed, where `list` is the most general type. In particular, integers and strings are considered as lists of length one. By abuse of terminology, we call items *unlabelled* if they are labelled with the empty list.

Besides a list label, nodes and edges may carry a *mark* which is one of the values `green`, `blue`, `grey` and `dashed` (where `grey` and `dashed` are reserved for nodes and edges, respectively). Marks are convenient to highlight items in input or output graphs, and to record which items have been visited during a graph traversal. For convenience, we sometimes refer to unmarked nodes as *white* nodes.

Moreover, nodes in rules and host graphs may be distinguished as *roots*. For example, in the rule `copy_root` of Fig. 2, nodes with a thick black border are roots. While roots are normally used to restrict the set of rule matches [1], we use them in this paper to specify reachable subgraphs.

The grammar in Fig. 5 of Appendix A gives the abstract syntax of graph programs in our subset (without the syntax of rule declarations). A program consists of a number of rule declarations and exactly one declaration of a main command sequence. The category RuleId refers to declarations of rules in RuleDecl. The call of a rule set $\{r_1, \ldots, r_n\}$ non-deterministically applies one of the rules whose

left-hand graph matches a subgraph of the host graph. Rule matching is injective and involves instantiating the variables in rules with host graph labels. The call *fails* if none of the rules is applicable to the host graph. A loop command $\mathcal{R}!$ applies the rule set $\mathcal{R}$ repeatedly until it fails. When this is the case, $\mathcal{R}!$ terminates with the graph resulting from the last successful application of $\mathcal{R}$.

The meaning of a graph program P is the function $[\![P]\!]$ mapping an input graph G to the set $[\![P]\!]G$ of all possible outcomes of executing P on G. Possible outcomes include the value fail which indicates a failed program run, and the value $\bot$ which indicates divergence. We say that program P *can diverge from* graph G if there exists an infinite program run starting from G.

Writing $\mathcal{G}^{\oplus}$ for the set of all host graphs extended with the values fail and $\bot$, the semantic function $[\![\_]\!]\colon \mathrm{ComSeq} \to (\mathcal{G} \to 2^{\mathcal{G}^{\oplus}})$ is defined by

$$[\![P]\!]G = \{X \in (\mathcal{G} \cup \{\mathrm{fail}\}) \mid \langle P, G\rangle \xrightarrow{+} X\} \cup \{\bot \mid P \text{ can diverge from } G\}$$

where $\to$ is the transition relation on configurations defined in Appendix B.

3 Verification of Graph Programs

As mentioned in the Introduction, we treat assertions semantically and express pre- and postconditions in ordinary mathematical language (similar to [10]). As is usual in Hoare logic, we use triples $\{c\}\ P\ \{d\}$ to state that program P is partially correct with respect to precondition c and postcondition d. Intuitively, this means that for every graph G satisfying c, any graph H resulting from executing P on G will satisfy d.

Given a graph G and some assertion c, we write $G \models c$ if G satisfies c. If, in addition to partial correctness, P cannot diverge or fail from graphs satisfying c, the program is totally correct.

Definition 1 (Partial and total correctness [14]). *A graph program P is* partially correct *with respect to a precondition c and a postcondition d, if for every host graph G and every graph H in $[\![P]\!]G$, $G \models c$ implies $H \models d$.*

P is totally correct *with respect c and d if it is partially correct and for every host graph G such that $G \models c$, $[\![P]\!]G \cap \{\bot, \mathrm{fail}\} = \emptyset$.*

We write $\models \{c\}\ P\ \{d\}$ if P is partially correct with respect to c and d. In Hoare logic, proof rules in the form of axioms and inference rules are used to construct proof trees decorated with Hoare triples. Proof trees are defined in Definition 8 of Appendix C. The rules we use in this paper are shown in Fig. 1; they are taken from [12, 13, 15] except for [rule app], which replaces an axiom involving the weakest liberal precondition. As our semantic assertions do not come with an algorithm for calculating the weakest liberal precondition, determining and proving this condition would unnecessarily inflate our proofs.

Property $\mathrm{App}(\mathcal{R})$ in rule [alap] expresses that the rule set $\mathcal{R}$ is applicable. By the semantics of the as-long-as-possible command, $\mathcal{R}$ is not applicable to any graph resulting from the loop $\mathcal{R}!$

$$\text{[ruleapp]}\ \frac{\models \{c\}\ r\ \{d\}}{\{c\}\ r\ \{d\}} \qquad \text{[ruleset]}\ \frac{\{c\}\ r\ \{d\} \text{ for all } r \in \mathcal{R}}{\{c\}\ \mathcal{R}\ \{d\}}$$

$$\text{[alap]}\ \frac{\{inv\}\ \mathcal{R}\ \{inv\}}{\{inv\}\ \mathcal{R}!\ \{\neg \mathrm{App}(\mathcal{R}) \wedge inv\}} \qquad \text{[comp]}\ \frac{\{c\}\ P\ \{d\} \quad \{d\}\ Q\ \{e\}}{\{c\}\ P;Q\ \{e\}}$$

$$\text{[cons]}\ \frac{c \Rightarrow c' \quad \{c'\}\ P\ \{d'\} \quad d' \Rightarrow d}{\{c\}\ P\ \{d\}}$$

Fig. 1. Proof rules for graph program verification

Definition 2 (App($\mathcal{R}$)). *Let $\mathcal{R}$ be a set of rules. A graph G satisfies* App($\mathcal{R}$) *if and only if there exists a graph H such that $G \Rightarrow_{\mathcal{R}} H$.*

The proof rules we use in this paper are known to be sound with respect to GP 2's operational semantics.

Theorem 1 (Soundness of proof rules [12]). *Given a program P and assertions c and d,*

$$\vdash \{c\}\ P\ \{d\} \text{ implies } \models \{c\}\ P\ \{d\}.$$

Here $\vdash \{c\}\ P\ \{d\}$ means that there exists a proof tree with root $\{c\}\ P\ \{d\}$.

4 Cheney's Copying Garbage Collector in GP 2

Cheney's garbage collector assumes two disjoint, equally large regions of memory where the first region holds the data structure to be garbage collected and the second region consists of free cells. The structure that is reachable from the root cell in the first area is copied to the second region. Subsequently, the complete first region can be freed by the memory management system. Adopting this technique, we construct the graph program `cheney` in Fig. 2 to garbage collect an input graph. We model the free cells assumed by Cheney's method as unlabelled isolated nodes. This is similar to the store model used by [6] for pointer verification.

As input, our program assumes a graph that can be partitioned into two subgraphs: the graph to be garbage collected, and the graph that models a region of free memory cells. We differentiate the two regions by colours. White and grey nodes are used for the first region while green and blue nodes are used for the second region. The root cell in the first region is a unique root node. Hence, garbage collection involves identifying the subgraph reachable from the root node and copying it to the unlabelled subgraph. In Fig. 3, we give an example of the execution of `cheney`.

As in [5, 16], we do not model the subsequent freeing of cells in the first region. This would be easy to achieve by a few rules which change all white and grey nodes into unlabelled green nodes and delete all edges between these nodes.

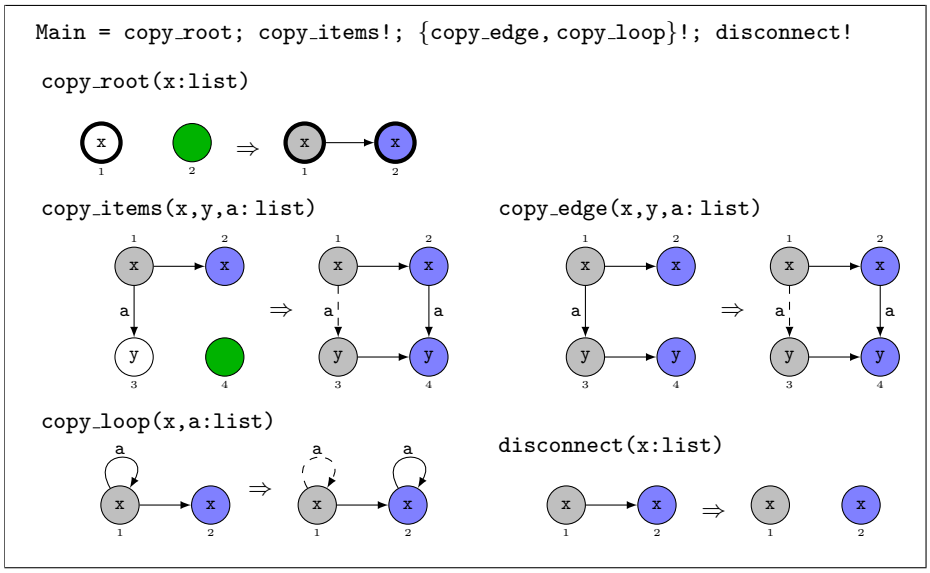

Fig. 2. Graph program `cheney`

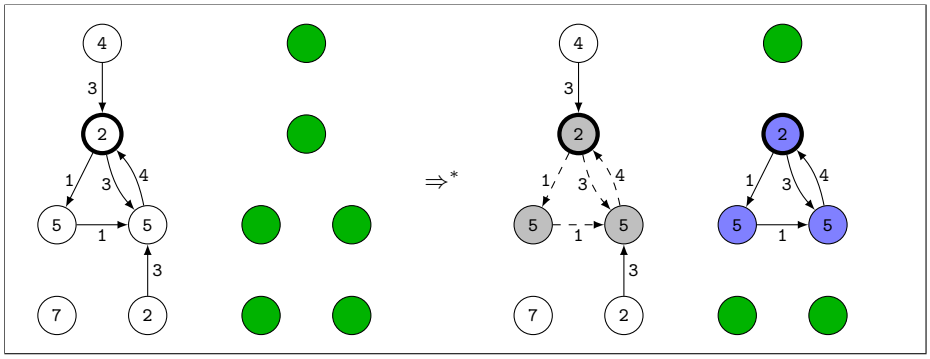

Fig. 3. A graph before and after the execution of `cheney`

5 Case Study: Specification

We assume that the input graph of the program `cheney` can be partitioned into two subgraphs as described above. This partition will persist during program execution. Given an input graph G, we use Old_G and New_G to refer to these subgraphs, where Old_G is to be garbage collected.

Definition 3 (Old_G **and** New_G)**.** *Given a graph G, we denote by Old_G the subgraph consisting of all unmarked and grey nodes and all edges between them.*

Also, New_G *denotes the subgraph consisting of all green and blue nodes and all edges between them.*

The aim of program `cheney` is to copy to New_G the subgraph of Old_G consisting of all nodes and edges that are reachable from the root. We denote this subgraph by $\text{Reach}(\text{Old}_G)$.

Definition 4 ($\text{Reach}(G)$**).** *Given a graph* G *with a unique root node* v*, we denote by* $\text{Reach}(G)$ *the subgraph of* G *consisting all nodes and edges reachable from* v *by directed paths.*

As the garbage collector copies a subgraph of Old_G to New_G, its formal specification by pre- and postcondition needs to require an isomorphism between Reach(Old) in the input graph and Reach(New) in the result graph. However, program `cheney` uses marks to distinguish Old and New and reachable subgraphs. Therefore, we introduce graph morphisms that preserve labels, sources, targets, and roots, but ignore marks.

Definition 5 (Liberal graph morphism). *Given graphs* G *and* H*, a* liberal graph morphism $f\colon G \to H$ *is a pair of mappings* $f = \langle f_V\colon V_G \to V_H,\ f_E\colon E_G \to E_H\rangle$ *that preserve sources, targets, labels and roots.*

Then, an isomorphism is a bijective liberal morphism that reflects root nodes.

Definition 6 (Isomorphism). *A liberal graph morphism* $f\colon G \to H$ *is an* isomorphism *if* f_V *and* f_E *are bijective and if for each node* v *in* G*,* v *is a root if and only if* $f_V(v)$ *is a root.*

Given an input graph `Start`, program `cheney` has to produce a graph `Result` such that $\text{Reach}(\text{Old}_{\texttt{Start}})$ is isomorphic to $\text{Reach}(\text{New}_{\texttt{Result}})$. We specify the pre- and postcondition of `cheney` as follows.

Precondition. Each node in `Start` is either unmarked or green, and the number of unmarked and green nodes is the same. All edges are unmarked. All green nodes are isolated and unlabelled. There is a unique root node which is unmarked.

Postcondition. Each node in `Result` is either in $\text{Old}_{\texttt{Result}}$ or $\text{New}_{\texttt{Result}}$, and the number of nodes in $\text{Old}_{\texttt{Result}}$ and $\text{New}_{\texttt{Result}}$ is the same. There are no edges connecting $\text{New}_{\texttt{Result}}$ and $\text{Old}_{\texttt{Result}}$. There is a unique grey root node in $\text{Old}_{\texttt{Result}}$ and a unique blue root node in $\text{New}_{\texttt{Result}}$. In $\text{Reach}(\text{Old}_{\texttt{Result}})$, the nodes are grey and the edges are dashed ,while other items in $\text{Old}_{\texttt{Result}}$ are unmarked. In $\text{Reach}(\text{New}_{\texttt{Result}})$, the nodes are blue and the edges are unmarked, while all other nodes in $\text{New}_{\texttt{Result}}$ are isolated green nodes which are unlabelled. Moreover, $\text{Reach}(\text{Old}_{\texttt{Start}})$ and $\text{Reach}(\text{New}_{\texttt{Result}})$ are isomorphic.

6 Case Study: Proof

6.1 Partial Correctness

To prove that program `cheney` is correct with respect to its pre- and postcondition, we consider an arbitrary execution of `cheney` that transforms an input graph `Start` satisfying the precondition into a result graph `Result`. Our proof rests on a property $\texttt{inv}_{\texttt{Start}}$, defined in Definition 7, which holds for `Start` and is an invariant for all five rules of `cheney` (shown in Lemma 2). Thus $\texttt{inv}_{\texttt{Start}}$ holds for each graph in the execution sequence $\texttt{Start} \Rightarrow^* \texttt{Result}$.

In particular, for each graph G such that $\texttt{Start} \Rightarrow^* G \Rightarrow^* \texttt{Result}$, $\texttt{inv}_{\texttt{Start}}$ asserts the existence of two isomorphisms: one between $\text{Old}_{\texttt{Start}}$ and Old_G and another one between Grey_G and Blue_G. Here Grey_G is the subgraph of G consisting of all grey nodes and all dashed edges between them, and Blue_G is the subgraph of G consisting all blue nodes and all edges between them. We then establish that $\text{Grey}_{\texttt{Result}}$ equals $\text{Reach}(\text{Old}_{\texttt{Result}})$ while $\text{Blue}_{\texttt{Result}}$ equals $\text{Reach}(\text{New}_{\texttt{Result}})$.

Roughly, $\text{Old}_{\texttt{Start}}$ and Old_G are isomorphic because (1) no rule deletes or creates nodes, (2) no rule deletes, creates or relabels edges within Old (any edge created or deleted is incident to a blue node), (3) no rule can change the colour or label of a grey node, and (4) rules can change unmarked nodes only by turning them grey (while preserving the label).

Moreover, Grey_G and Blue_G are isomorphic because (1) $\text{Grey}_{\texttt{Start}} = \emptyset = \text{Blue}_{\texttt{Start}}$, (2) `copy_root` creates one-node graphs Grey and Blue whose nodes are roots with the same label, and an unlabelled edge connecting Grey and Blue, called an *isomorphism edge*, which represents the node mapping of the isomorphism, (3) `copy_items` extends both Grey and Blue by one edge and its target node, where the edges have the same label and sources connected by an isomorphism edge, and creates an isomorphism edge between the target nodes, (4) `copy_edge` extends Grey and Blue by one edge each, where the edges have the same label and have their sources resp. targets connected by an isomorphism edge, (5) `copy_loop` works similar to `copy_edge` except that the new edges are loops, and (6) `disconnect` removes an isomorphism edge and hence does not alter Grey or Blue.

We use these isomorphisms to show that $\text{Reach}(\text{Old}_{\texttt{Start}})$, $\text{Reach}(\text{Old}_{\texttt{Result}})$ and $\text{Reach}(\text{New}_{\texttt{Result}})$ are all isomorphic, thus establishing the correctness of the garbage collector. We remark that verifying the existence of an isomorphism between (subgraphs of) the start graph and the result graph of a graph program execution is not possible with the approach of [13, 14, 12, 15].

A proof tree for the partial correctness for `cheney` is provided in Fig. 4. The assertions in the tree are defined in Definition 7. One of these assertions is $\texttt{inv}_{\texttt{Start}}$ which acts as an invariant of `cheney` (see Lemma 2). We then give arguments about leaves in the proof tree in Lemma 3. The proof tree contains some applications of the proof rule [cons] whose validity is obvious by propositional logic, such as $c \wedge d \Rightarrow c$. We do not justify such applications, but we prove in Lemma 4 implications that are not obvious.

Definition 7. *Let G be a graph. We define the following assertions:*

$\texttt{inv}_{\texttt{Start}}$	:	(a) every node in G is either in Old_G or New_G, where Old_G and New_G have the same number of nodes (b) there is a unique root node in Old_G and at most one root node in New_G (c) each edge in G is either unmarked or dashed (d) all grey nodes are in $\text{Reach}(\text{Old}_G)$ (e) all dashed edges are in $\text{Reach}(\text{Old}_G)$ (f) all blue nodes are in $\text{Reach}(\text{New}_G)$ (g) all green nodes are isolated unlabelled nodes (h) there exists an isomorphism $f\colon \text{Grey}_G \to \text{Blue}_G$ where Grey_G is the subgraph of G consisting all grey nodes and all dashed edges between them while Blue_G is the subgraph of G consisting all blue nodes and all edges between them (k) each edge e connecting Old_G and New_G is an isomorphism edge, that is, an unlabelled and unmarked edge satisfying $f_V(s_G(e)) = t_G(e)$ (l) Old_G is isomorphic to $\text{Old}_{\texttt{Start}}$
`reachOldGrey`	:	in $\text{Reach}(\text{Old}_G)$, all nodes are grey
`reachOldDashed`	:	in $\text{Reach}(\text{Old}_G)$, all edges are dashed
`reachNewBlue`	:	in $\text{Reach}(\text{New}_G)$, all nodes are blue
`nogreynode`	:	there is no grey node
`nobluenode`	:	there is no blue node
`nodashededge`	:	there is no dashed edge
`noconnect`	:	there are no edges between Old_G and New_G
`connect`	:	if $f_V(v_1) = v_2$ then there exists an edge from v_1 to v_2
`rootOldUnmark`	:	there exists an unmarked root node in Old_G
`rootOldGrey`	:	there exists a grey root node in Old_G
`norootNew`	:	there is no root node in New_G
`rootNewBlue`	:	there exists a blue root node in New_G

From now on we denote the pre- and postcondition stated in Section 5 by `precondition` and `postcondition`, respectively.

Lemma 1 (Pre- and postcondition). *Using the assertions of* Definition 7*, the following holds:*

$$\texttt{precondition} \Leftrightarrow \texttt{inv}_{\texttt{Start}}(\texttt{a},\texttt{b},\texttt{c},\texttt{g}) \wedge \texttt{nogreynode} \wedge \texttt{nobluenode} \wedge \texttt{nodashededge} \wedge \texttt{rootOldUnmark} \wedge \texttt{norootNew}$$

$$\texttt{postcondition} \Leftrightarrow \texttt{inv}_{\texttt{Start}}(\texttt{a},\texttt{b},\texttt{d},\texttt{e},\texttt{f},\texttt{g},\texttt{h},\texttt{l}) \wedge \texttt{noconnect} \wedge \texttt{reachOldGrey} \wedge \texttt{reachOldDashed} \wedge \texttt{reachNewBlue}$$

Proof. The first sentence of `precondition` is equivalent to $\texttt{inv}_{\texttt{Start}}(\texttt{a}) \wedge$`nogreynode` $\wedge$`nobluenode`, while the second sentence equivalent to $\texttt{inv}_{\texttt{Start}}(\texttt{c}) \wedge$`nodashededge`. Then, the next sentence is equivalent to $\texttt{inv}_{\texttt{Start}}(\texttt{g})$, and the last sentence is equivalent to $\texttt{inv}_{\texttt{Start}}(\texttt{b}) \wedge$ `rootOldUnmark` $\wedge$ `norootNew`.

For `postcondition`, note that we write the result graph as `Result`. The first sentence in `postcondition` is equivalent to $\texttt{inv}_{\texttt{Start}}(\texttt{a})$, while the second sentence is equivalent to `noconnect`. Then, the next sentence is equivalent to $\texttt{inv}_{\texttt{Start}}(\texttt{b}) \wedge \texttt{rootOldGrey} \wedge \texttt{rootNewBlue}$. With the support of $\texttt{inv}_{\texttt{Start}}(\texttt{a})$, the fourth sentence is equivalent to $\texttt{reachOldGrey} \wedge \texttt{reachOldDashed} \wedge \texttt{inv}_{\texttt{Start}}(\texttt{d}) \wedge \texttt{inv}_{\texttt{start}}(\texttt{e})$, and the fifth sentences is equivalent to $\texttt{reachNewBlue} \wedge \texttt{inv}_{\texttt{Start}}(\texttt{f}) \wedge \texttt{inv}_{\texttt{Start}}(\texttt{g})$. The last sentence then equivalent to $\texttt{inv}_{\texttt{Start}}(\texttt{h}) \wedge \texttt{inv}_{\texttt{Start}}(\texttt{l})$. □

Lemma 2 ($\texttt{inv}_{\texttt{Start}}$ is invariant for all rules). *The following holds:*

$\models \{\texttt{inv}_{\texttt{Start}}\}$ `copy_root` $\{\texttt{inv}_{\texttt{Start}}\}$ $\quad\quad$ $\models \{\texttt{inv}_{\texttt{Start}}\}$ `copy_loop` $\{\texttt{inv}_{\texttt{Start}}\}$
$\models \{\texttt{inv}_{\texttt{Start}}\}$ `copy_items` $\{\texttt{inv}_{\texttt{Start}}\}$ $\quad\quad$ $\models \{\texttt{inv}_{\texttt{Start}}\}$ `disconnect` $\{\texttt{inv}_{\texttt{Start}}\}$
$\models \{\texttt{inv}_{\texttt{Start}}\}$ `copy_edge` $\{\texttt{inv}_{\texttt{Start}}\}$

Proof. We proof the lemma by checking each point of $\texttt{inv}_{\texttt{Start}}$.

(a) The application of any rule in `cheney` does not add or remove any node. It may change the colour of a node from unmarked to grey or from green to blue, which does not change the number of vertices in Old_G and New_G.

(b) Changes in root node only shown in `copy_root`, which change unmarked root node to grey root node and add a blue root node. This means the number of root nodes in Old_G never change after any rule application, while the number of root nodes in New_G can increase to one after the application of `(copy_root)`, but never change after the application of other rules.

(c) Every edge in each rule in the lemma is either unmarked or dashed. Therefore, for all rules in the lemma, if there is no other marks for edges exists before a rule application, they must not exist after the rule application.

(d) We only give the proof for `copy_items` and `copy_edge` as the other rules do not change grey nodes and the triples must hold. For `copy_root`, the rule change an unmarked root node to a grey root node. From (b) we know that there is only one root node in Old_G, which implies there is no other root node than this new grey root node. The new grey root node is reachable from itself. Then for `copy_items`, the rule change an unmarked node, which is reachable from a grey node, to grey. This means the new grey node is reachable from the root node as well.

(e) We can see that in every rule in the lemma, if there exists a dashed edge then its source and target must be grey. Since grey nodes are in $\text{Reach}(G)$, dashed edges must also be in $\text{Reach}(\text{Old}_G)$.

(f) Similar to the proof for (d).

(g) There is no rule in the lemma transform an isolated unlabelled green node into a green node that incident to an edge or not unlabelled.

(h) In every rule in the lemma, a production of a new grey node is always accompanied with a production of a new blue node. This also holds for dashed edge and an edge between blue nodes. Then, the new grey node can map to the new blue node, and the new dashed edge can map to the new edge as an addition to the morphism f. The mappings are isomorphism as they preserve sources, targets, labels, and root.

$$\dfrac{\dfrac{\text{subtree I} \qquad \text{subtree II}}{\{\text{X1}\}\ \texttt{copy\_root; copy\_item!; \{copy\_edge, copy\_loop\}!; disconnect!}\ \{\text{Y1}\}}\ [\text{comp}]}{\{\texttt{precondition}\}\ \texttt{Main}\ \{\texttt{postcondition}\}}\ [\text{cons}]$$

where
$\text{X1} = \texttt{inv}_{\texttt{Start}} \wedge \texttt{rootOldUnmark} \wedge \texttt{norootNew} \wedge \texttt{nogreynode} \wedge \texttt{nodashededge} \wedge \texttt{nobluenode} \wedge \texttt{noconnect}$,
$\text{Y1} = \texttt{inv}_{\texttt{Start}} \wedge \texttt{rootOldUnmark} \wedge \texttt{rootNewBlue} \wedge \texttt{reachOldGrey} \wedge \texttt{reachOldDashed} \wedge \texttt{reachNewBlue} \wedge \texttt{noconnect}$,

subtree I is:

$$\dfrac{\dfrac{\dfrac{}{\{\text{X2}\}\texttt{copy\_root}\{\text{X3}\}}\ [\text{ruleapp}]}{\{\text{X1}\}\texttt{copy\_root}\{\text{X3}\}}\ [\text{cons}] \qquad \dfrac{\dfrac{\dfrac{}{\{\text{X4}\}\texttt{copy\_items}\{\text{X4}\}}\ [\text{ruleapp}]}{\{\text{X4}\}\texttt{copy\_items!}\{\text{X4} \wedge \neg\text{App(copy\_items)}\}}\ [\text{alap}]}{\{\text{X3}\}\texttt{copy\_items!}\{\text{X5}\}}\ [\text{cons}]}{\{\text{X1}\}\texttt{copy\_root; copy\_items!}\{\text{X5}\}}\ [\text{comp}]$$

where $\text{X2} = \texttt{inv}_{\texttt{Start}} \wedge \texttt{rootOldUnmark} \wedge \texttt{norootNew} \wedge \texttt{nodashededge} \wedge \texttt{noconnect}$,
$\text{X3} = \texttt{inv}_{\texttt{Start}} \wedge \texttt{rootOldGrey} \wedge \texttt{rootNewBlue} \wedge \texttt{reachNewBlue} \wedge \texttt{nodashededge} \wedge \texttt{connect}$,
$\text{X4} = \texttt{inv}_{\texttt{Start}} \wedge \texttt{rootOldGrey} \wedge \texttt{rootNewBlue} \wedge \texttt{reachNewBlue} \wedge \texttt{connect}$,
$\text{X5} = \text{X4} \wedge \texttt{reachOldGrey}$,

and subtree II is:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{\text{X5}\}\texttt{copy\_edge}\{\text{X5}\}}\ [\text{ruleapp}] \qquad \dfrac{}{\{\text{X5}\}\texttt{copy\_loop}\{\text{X5}\}}\ [\text{ruleapp}]}{\{\text{X5}\}\{\texttt{copy\_edge,copy\_loop}\}\{\text{X5}\}}\ [\text{ruleset}]}{\{\text{X5}\}\{\texttt{copy\_edge,copy\_loop}\}\texttt{!}\{\text{X5} \wedge \neg\text{App}(\{\texttt{copy\_edge,copy\_loop}\})\}}\ [\text{alap}]}{\{\text{X5}\}\{\texttt{copy\_edge,copy\_loop}\}\texttt{!}\{\text{X6}\}}\ [\text{cons}] \qquad \dfrac{\dfrac{\dfrac{}{\{\text{X7}\}\texttt{disconnect}\{\text{X7}\}}\ [\text{ruleapp}]}{\{\text{X7}\}\texttt{disconnect!}\{\text{X7} \wedge \neg\text{App(disconnect)}\}}\ [\text{alap}]}{\{\text{X6}\}\texttt{disconnect!}\{\text{Y1}\}}\ [\text{cons}]}{\{\text{X5}\}\{\texttt{copy\_edge,copy\_loop}\}\texttt{!; disconnect!}\{\text{Y1}\}}\ [\text{comp}]$$

where $\text{X6} = \text{X5} \wedge \texttt{reachOldDashed}$,
$\text{X7} = \texttt{inv}_{\texttt{Start}} \wedge \texttt{rootOldGrey} \wedge \texttt{rootNewBlue} \wedge \texttt{reachNewBlue} \wedge \texttt{reachOldGrey} \wedge \texttt{reachOldDashed}$,

Fig. 4. Proof tree for partial correctness of `cheney`

(k) The edge between Old and New can be inserted only by rules `copy_root` and `copy_items`. This means that the edge must be an isomorphism edge.

(l) Every rule in the lemma does not add or delete any unmarked or grey nodes, or edges between them. Modification of these elements exists, where unmarked nodes can transform into grey nodes and unmarked edges can transform into dashed edges while labels are still preserved. However, the marks indicate no changes in area (New or Old) membership, and there are no changes in adjacency. Hence, sources, targets, and labels are preserved. □

Lemma 3. *For assertions* X2, X3, X4, X5, X7 *as defined in* Fig. 4*, the following holds:*

(1) $\models$ {X2} `copy_root` {X3}
(2) $\models$ {X4} `copy_items` {X4}
(3) $\models$ {X5} `copy_edge` {X5}
(4) $\models$ {X5} `copy_loop` {X5}
(5) $\models$ {X7} `disconnect` {X7}

Proof. Recall that by Lemma 2, each rule preserves $\texttt{inv}_{\texttt{Start}}$.

(1) We can see that the application of `copy_root` preserves the satisfiability of `nodashededge`. The assertion `rootOldUnmark` in the precondition guarantees `rootOldGrey` by the construction of `copy_root`. Then because the rule creates a new rooted node in the New subgraph, `norootNew` yields `rootNewBlue` after the rule application. `connect` must holds because the new edge must be the only one connecting the two areas as `noconnect` holds in the precondition. It also guarantees `reachNewBlue` because in New subgraph, the new blue root node is an isolated node, so it is the only node that is reachable from the root node.

(2) We can see that `copy_items` does not change any root, grey, or blue node, so it preserves `rootOldGrey` $\wedge$ `rootNewBlue` as well. Then, `connect` asserts there exists an edge between v_1 and v_2 for all v_1 and v_2 such that $f_v(v_1) = v_2$. From $\texttt{inv}_{\texttt{Start}}$, we know that v_1 is a grey node and v_2 is a blue node. `copy_items` yields a new grey and blue node with the same label and there is an edge between them. Hence, `connect` is preserved. The rule `copy_items` also changes the number of blue node, so we will need to see how it affects `reachNewBlue`. Follows from point (f) of $\texttt{inv}_{\texttt{Start}}$, the blue node in the left-hand side of `copy_items` must be in $\texttt{Reach}(\texttt{New}_{\texttt{G}})$. Therefore, the new blue node in the right-hand side is in $\texttt{Reach}(\texttt{New}_{\texttt{G}})$ as well, so that `reachNewBlue` still holds.

(3) Because there are no changes in nodes, `copy_edge` preserves `rootOldGrey` $\wedge$ `rootNewBlue`. The adjacency between two grey nodes and between a grey and a blue node are not changed by `copy_edge`, so `connect` $\wedge$ `reachOldGrey` still holds. There is a change in adjacency between two blue nodes, but it still preserves `reachNewBlue` as if the change makes $\texttt{Reach}(\texttt{New}_{\texttt{G}})$ gets an additional node, it will be a blue node.

(4) The rule does not add or remove any node, also any edge between two nodes, so it preserves `rootOldGrey` $\wedge$ `rootNewBlue` $\wedge$ `reachNewBlue` $\wedge$ `reachNewGrey` $\wedge$ `connect`.

(5) Because `disconnect` does not change any node, also any edge between same-coloured nodes, it preserves `rootOldGrey`$\wedge$`rootNewBlue`$\wedge$`reachNewBlue`$\wedge$ `reachOldGrey`$\wedge$`reachOldDashed`. □

Lemma 4 (Validity of implications). *For assertions* X1, X4, X5, X6, X7, Y1 *as defined in* Fig. 4*, the following holds:*

(1) `precondition` $\Rightarrow$ X1
(2) Y1 $\Rightarrow$ `postcondition`
(3) X5 $\wedge \neg$App(`copy_edge`) $\Rightarrow$ X6
(4) X5 $\wedge \neg$App(`copy_loop`) $\Rightarrow$ X6
(5) X4 $\wedge \neg$App(`copy_items`) $\Rightarrow$ X5
(6) X7 $\wedge \neg$App(`disconnect`) $\Rightarrow$ Y1

Proof. (1) We will show that `precondition` in Lemma 1 implies X1. In X1, we have additional conjunction point (d), (e), (f), (h), (k), and (l) of $\texttt{inv}_{\texttt{Start}}$. Point (l) is clearly satisfied as `G` is `Start`. Then the other points must hold because `nogreynode` $\wedge$ `nobluenode` $\wedge$ `nodashedge` implies the nodes or edges that must satisfies certain requirement do not exist so nothing negate those points. Then, `noconnect` holds because New area only consists of isolated nodes.

(2) By simplification, it is clear that Y1 implies `postcondition`.

(3, 4) Similar as above, the non-applicability of `copy_edge` and `copy_loop` implies that there is no unmarked edges (including loops) where its source and target is a grey node. Therefore, `reachOldGrey` implies all nodes in Reach(Old_G) are grey and the non-applicability implies that edges between these nodes are not unmarked, i.e. all edges between these nodes are dashed, `reachOldDashed` holds. Hence, X6 holds.

(5) `connect` and isomorphism (h) in $\texttt{inv}_{\texttt{Start}}$ assert that each grey node is a source for an edge where a blue node with the same label as the grey node is the target. `rootOldGrey` implies the existence of grey node. $\texttt{inv}_{\texttt{Start}}$ also implies that if there exists an unmarked node, then there must exist an isolated node, as the number of grey and unmarked nodes equal to the number of blue and isolated green nodes while there is a bijective function from grey nodes to blue nodes. Therefore non-applicability of `copy_items` implies that there is no edge from a grey node to an unmarked node, which means unmarked nodes are not reachable from grey nodes. Then because $\texttt{inv}_{\texttt{Start}}$ asserts grey nodes are reachable from the start node in Old area, the unmarked nodes must not reachable from the start node so that `reachOldGrey` holds. Hence, X5 holds.

(6) Non-applicability of `disconnect` implies there are no edges between any grey nodes and any blue nodes. $\texttt{inv}_{\texttt{Start}}$ implies that edges connecting Old_G and New_G are only edges incident to grey and blue nodes. Therefore, the non-applicability implies Old_G and New_G are not connected which means `noconnect` holds so that Y1 holds. □

6.2 Total Correctness

A graph program P is totally correct with respect to a precondition c if the graph program is partially correct, also will not fail or diverge, with respect to c [14]. We have shown that the program `cheney` is partially correct, so we only need to show that `cheney` cannot fail or diverge.

Let us recall `cheney` at Fig. 2. The command sequence in the program consists of one rule set call and four loop commands. `Precondition` clearly stated the existence of an unmarked root node. Then, because `precondition` guarantees

the same number of unmarked and green nodes, there must exists at least one green node as we have an unmarked root node. Therefore, the match of the left-hand side of `copy_root` is guaranteed by `precondition` so it will not fail. By the semantic of loop command, the other commands in the sequence will not fail either so that the absence of failure in the execution of `cheney` is guaranteed.

The rule `copy_root` is only applied once, so it will not diverge. For loop commands, elimination of an element is clearly can guarantee the absence of divergence. The rule `copy_items, copy_edge, copy_loop`, and `disconnect` eliminate unmarked nodes, unmarked edges between two grey nodes, unmarked loops on grey nodes, and edges between grey and blue nodes repectively.

7 Related Work

In this study, we implement Cheney's copying garbage collector [3] in GP 2 and reason about it. Several works in verification of Cheney's algorithm stated the difficulties in verifying the algorithm, such as in reasoning about reachability in graphs [8] and verifying programs involving cyclic data structures.

Torp-Smith et.al. [16] is the extended version of [2]. In the study, they extend standard separation logic so that it can be used in cyclic data structures. Some remarks are stated to show the advantage in using local reasoning for the verification, such as ensuring an assignment is not affecting some assertions. The isomorphism between data structures from two different points in time; before and after a program execution; is also introduced in the study. Again, separating conjunction is used to reason about isomorphism as the update of bijective function can be seen in local reasoning. However, we think their proof is complicated as there are so many rules involved yet the validity of the rules is not provided. Moreover, there are 57 pages in the journal paper to discuss the verification of Cheney's garbage collector. They use about 48 triples in the proof, but reasoning about each step takes a lot of work as well.

In McCreight's PhD thesis [7], they use the definition of a morphism from [16] and study about mechanised verification of Cheney's algorithm. They also use separation logic for their verification. Their proof is detailed as they see all possible cases to mechanise the verification, but they use various lemmas, and it is not clear how the lemmas are proved. The proof is separated into five parts. The verification of each part requires between one and seven pages. However, to fully understand the verification, we need to understand the specification of each part which is not more concise than the verification itself. In total, there are about 50 pages of the thesis that deal with Cheney's algorithm.

This may result from the level difference in the language, as they use low-level programming and we use a programming language that abstracts from details of memory management such as address arithmetic. The other studies use separation logic for local reasoning in the verification. However, because sharing mostly exists in graph problems including garbage collector, they need to extend separation logic so that it can be used in the verification of Cheney's algorithm. The extension itself is not easy, and proofs following this extension

is complex. In contrast, there is no need for us to extend the existing proof calculus for reasoning about our graph program. By using the existing proof calculus, which is sound in the sense of partial correctness, we are able to show verification of `cheney` in a simple way with clear justification for each step of verification.

Another extension of separation logic for the verification of Cheney's garbage collector can be found in [5]. The paper introduces the notion of sharing in separation logic, which is called ramification. This allows local reasoning while global effects are still accounted for when they are required, enabling reasoning about programs that manipulate data structures with unrestricted sharing. Different from [2, 16], the paper uses inductive graph predicates and does the reasoning on the level of mathematical graphs. It is claimed that the verification is more concise than in other work. This is indeed the case as there are only about two pages to discuss Cheney's garbage collector and its verification. However, we find it difficult to see the reasoning about implications given in the proof. In addition, one needs to understand the intricate theory about ramification and its use in verification.

8 Conclusion and Future Work

We have implemented Cheney's copying garbage collector in GP 2 and verified the program using Hoare logic. To be compared to the previous work we described in the previous section, they use local reasoning and argue that this helps them so that their reasoning is less complicated [16]. To be compared with our work, the use of marks in our program implicitly helps us in separating properties that are not affected as we can focus on structures with specific marks. Similarly, the update of bijective function in our case can be seen with the use of marks.

We show a proof tree for the partial correctness of the program, and only from this, we argue that the proof is more straightforward than other proofs that have been done for the Cheney's algorithm. Moreover, in our work we use proof rules that are proven to be sound to connect one triple to another. But in other literature about verification of Cheney's algorithm, they do not have a clear reasoning about soundness of proof rules they use. Moreover, from the proof sketch, we only use 22 triples while the proof sketch in [16] has about 48 triples. This shows how concise our proof if we compare to theirs.

Our proof is more concise than others partly because we still use arbitrary mathematical language for the assertions, unlike others that have defined formal assertions for this. Although there are E-conditions [12] that can be used to express properties of graphs in graph programs, we still need to extend this. We need an assertion that can describe a condition between two graphs that exist at a different point in time, but E-condition we have now only expressed the graph that exists at one point in time. Moreover, E-condition that based on first-order logic is not enough to represent properties we need, e.g. the existence of two area Old and New. M-condition [15] can cover this, but the formal definition of this is

yet to be defined. However, none of these can be used to express the isomorphism between two graphs that exist in different time. In the future, we plan to look the transduction in monadic second-order logic [4] and have assertion language that can express isomorphism between two structures.

References

1. C. Bak and D. Plump. Compiling graph programs to C. In *Proc. International Conference on Graph Transformation (ICGT 2016)*, volume 9761 of *Lecture Notes in Computer Science*, pages 102–117. Springer, 2016.
2. L. Birkedal, N. Torp-Smith, and J. C. Reynolds. Local reasoning about a copying garbage collector. In *Proc. Symposium on Principles of Programming Languages (POPL 2004)*, pages 220–231. ACM, 2004.
3. C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, 1970.
4. B. Courcelle and J. Engelfriet. *Graph structure and monadic second-order logic: a language-theoretic approach*. Cambridge University Press, 2012.
5. A. Hobor and J. Villard. The ramifications of sharing in data structures. In *Proc. Symposium on Principles of Programming Languages (POPL 2013)*, pages 523–536. ACM, 2013.
6. N. Klarlund and M. Schwartzbach. Verification of pointers. DAIMI Report Series 23(470), Aarhus University, 1994.
7. A. E. Mccreight. *The Mechanized Verification of Garbage Collector Implementations*. PhD thesis, Yale University, 2008.
8. M. O. Myreen. Reusable verification of a copying collector. In *Proc. Verified Software: Theories, Tools, Experiments (VSTTE 2010)*, volume 6217 of *Lecture Notes in Computer Science*. Springer, 2010.
9. G. D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139, 2004.
10. D. Plump. Reasoning about graph programs. In *Proc. Computing with Terms and Graphs (TERMGRAPH 2016)*, volume 225 of *Electronic Proceedings in Theoretical Computer Science*, pages 35–44, 2016.
11. D. Plump. From imperative to rule-based graph programs. *Journal of Logical and Algebraic Methods in Programming*, 88:154–173, 2017.
12. C. M. Poskitt. *Verification of Graph Programs*. PhD thesis, University of York, 2013.
13. C. M. Poskitt and D. Plump. Hoare-style verification of graph programs. *Fundamenta Informaticae*, 118(1-2):135–175, 2012.
14. C. M. Poskitt and D. Plump. Verifying total correctness of graph programs. In *Proc. International Workshop on Graph Computation Models (GCM 2012)*, 2012. Revised version in Volume 61 of Electronic Communications of the EASST, 2013.
15. C. M. Poskitt and D. Plump. Verifying monadic second-order properties of graph programs. In *Proc. International Conference on Graph Transformation (ICGT 2014)*, volume 8571 of *LNCS*, pages 33–48. Springer, 2014.
16. N. Torp-Smith, L. Birkedal, and J. C. Reynolds. Local reasoning about a copying garbage collector. *ACM Transactions on Programming Languages and Systems*, 30(4):24:1–24:58, 2008.

A Abstract Syntax of Graph Programs

In Figure 5 we give the syntax of the subset of GP 2 programs that we use in this paper.

```
Prog        ::= Decl {Decl}
Decl        ::= RuleDecl | MainDecl
MainDecl    ::= Main '=' ComSeq
ComSeq      ::= Com {';' Com}
Com         ::= RuleSetCall ['!']
RuleSetCall ::= RuleId | '{' RuleId {',' RuleId} '}'
```

Fig. 5. Abstract syntax of a subset of GP 2 programs

B Operational Semantics of Graph Programs

This appendix reviews the semantics of graph programs (except for the definition of rule applications), which is given in the style of structural operational semantics [9]. In this approach, inference rules inductively define a small-step transition relation $\rightarrow$ on *configurations*. In our setting, a configuration is either a command sequence together with a host graph, just a host graph or the special element fail:

$$\rightarrow \subseteq (\text{ComSeq} \times \mathcal{G}) \times ((\text{ComSeq} \times \mathcal{G}) \cup \mathcal{G} \cup \{\text{fail}\}).$$

Configurations in $\text{ComSeq} \times \mathcal{G}$, given by a rest program and a host graph, represent states of unfinished computations while graphs in $\mathcal{G}$ are final states or *results* of computations.

Figure 6 shows the inference rules for the GP 2 commands used in this paper. The rules contain meta-variables for command sequences and graphs, where R stands for a call in category RuleSetCall (as defined by the grammar in Figure 5), P and Q stand for command sequences in category ComSeq, and G, H stand for graphs in $\mathcal{G}$. The transitive closure of $\rightarrow$ is denoted by $\rightarrow^+$. We write $G \Rightarrow_R H$ if H results from host graph G by applying the rule set R, while $G \not\Rightarrow_R$ means that there is no graph H such that $G \Rightarrow_R H$ (application of R fails).

C Proof Trees

Proof trees are inductively defines as follows.

Definition 8 (Proof tree [12]). *If* $\{c\}\ P\ \{d\}$ *is an instance of an axiom* X *then*

$$X\ \frac{}{\{c\}\ P\ \{d\}}$$

$$[\text{call}_1]\ \frac{G \Rightarrow_R H}{\langle R,\, G\rangle \to H} \qquad [\text{call}_2]\ \frac{G \not\Rightarrow_R}{\langle R,\, G\rangle \to \text{fail}}$$

$$[\text{seq}_1]\ \frac{\langle P,\, G\rangle \to \langle P',\, H\rangle}{\langle P; Q,\, G\rangle \to \langle P'; Q,\, H\rangle} \quad [\text{seq}_2]\ \frac{\langle P,\, G\rangle \to H}{\langle P; Q,\, G\rangle \to \langle Q,\, H\rangle}$$

$$[\text{seq}_3]\ \frac{\langle P,\, G\rangle \to \text{fail}}{\langle P; Q,\, G\rangle \to \text{fail}}$$

$$[\text{alap}_1]\ \frac{\langle P,\, G\rangle \to^+ H}{\langle P!,\, G\rangle \to \langle P!,\, H\rangle} \qquad [\text{alap}_2]\ \frac{\langle P,\, G\rangle \to^+ \text{fail}}{\langle P!,\, G\rangle \to G}$$

Fig. 6. Semantic inference rules

is a proof tree. If $\{c\}\ P\ \{d\}$ *can be instantiated from the conclusion of an inference rule* X*, and there are proof trees* $T_1, \cdots, T_n$ *with conclusions that are instances of the* n *premises of* X*, then*

$$X\ \frac{T_1 \quad \cdots \quad T_n}{\{c\}\ P\ \{d\}}$$

is a proof tree.