Automated Verification of Safety Properties of Declarative Networking Programs

Chen Chen, Lay Kuan Loh, Limin Jia, Wenchao Zhou, Boon Thau Loo

July 1, 2015

(Updated December 1, 2015)

CMU-CyLab-15-002

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213
Automated Verification of Safety Properties of Declarative Networking Programs

(Extended Technical Report)

Chen Chen
University Of Pennsylvania
chenche@seas.upenn.edu

Lay Kuan Loh
Carnegie Mellon University
lkloh@cmu.edu

Limin Jia
Carnegie Mellon University
liminjia@cmu.edu

Wenchao Zhou
Georgetown University
wzhou@cs.georgetown.edu

Boon Thau Loo
University Of Pennsylvania
boonloo@cis.upenn.edu

Abstract

Networks are complex systems that unfortunately are ridden with errors. Such errors can lead to disruption of services, which may have grave consequences. Verification of networks is key to eliminating errors and building robust networks. In this paper, we propose an approach to verify networks using declarative networking, where networks are specified in NDLog, a declarative language. We focus on analyzing safety properties. We develop a technique to statically analyze NDLog programs: first, we build a dependency graph of the predicates of NDLog programs; then, we build a summary data structure called a derivation pool to represent all possible derivations and their associated constraints for predicates in the program; finally, properties specified in first-order logic are checked on the summary data structure with the help of the SMT solver Z3. We build a prototype tool and demonstrate the effectiveness of the tool in validating and debugging several SDN applications.

Keywords

Declarative networking, static analysis

1. Introduction

As more and more services are offered over the Internet, ensuring the security and stability of networks has become increasingly important. Unfortunately, networks are complex systems that are ridden with errors. Such errors can lead to disruption of services, which may have grave consequences. Verification of networks is key to eliminating errors and building robust networks. Much work on network verification has focused on verifying topological-specific network configurations [17, 21, 32, 37]. Practical testing tools for finding undesired behavior in protocol implementation have also been proposed [15, 24]. With the emerging technology of software-defined networking (SDN), modeling networks as programmable software has gained unprecedented popularity. Researchers began to apply program verification techniques to the verification of SDNs [8, 9].

Our goal is to develop a general automated technique that can be applied to network verification. The first step towards that goal is to find the right abstraction for networks. Declarative networking [30] is one of the first research efforts to demonstrate that high-level languages can be used to program networks. In declarative networking, network protocols are written in a declarative language NDLog, which is a distributed Datalog. Declarative networking techniques have been used in several domains including fault tolerance, protocols [44], cloud computing [4], sensor networks [12], overlay network compositions [33], anonymity systems [43], mobile ad-hoc networks [26, 35], wireless channel selection [25], network configuration management [11], and forensic analysis [52–54]. An open-source declarative networking system called RapidNet [42] has been integrated with the ns-3 [38] simulator, so protocols can be tested. It has also been shown that network verification can be carried out using the declarative network framework [10, 46, 47]. In summary, NDLog is a great intermediary language for bridging the gap between network specification, verification, and implementation, so we use NDLog as our specification language for networks.

Unfortunately, all of the verification tools related to NDLog require manual proofs, which makes verification very labor intensive. What is worse is that when the proofs cannot be constructed, it is nontrivial to find out what went wrong. Either there are bugs in the program, or the invariants used in the proofs are not correct. There is little tool support for identifying problems under these circumstances. In this paper, we develop an automated static analysis technique to analyze the safety properties of NDLog programs. When properties do not hold, our tool provides a concrete counterexample to further aid program debugging. The properties that we are interested in include invariants of the network and desirable behavior of nodes in the network. For instance, we would like to know if every forward entry corresponds to a route announcement packet, or if a successfully delivered packet indicates proper forwarding table setup in the switches that the packet traverses. One observation we have is that a large fragment of the interesting properties of networks can be expressed in a simple fragment of first-order logic. Leveraging this limited expressive power, we are able to develop static analysis for NDLog programs.

Our static analysis examines the structure of the NDLog program and builds a summary data structure for all derivations of that program. Properties specified in the restricted format of first-order logic are checked on the summary data structure with the help of the SMT solver Z3 [49]. The challenge is how to deal with recursive programs. For such programs, the number of possible derivations for recursive predicates is infinite. We use a concise representation for recursive predicates, so all possible derivations can be finitely represented. To evaluate our analysis, we built a prototype tool, and verified several safety properties of a number of SDN controller programs, where the SDN’s controller program and switch logic are specified in NDLog.
This paper makes the following technical contributions.

- We developed algorithms for automatically analyzing a class of safety properties of NDLog programs.
- We proved the correctness (soundness and completeness) of our algorithms for non-recursive programs and proved the soundness of our algorithms for recursive programs.
- We implemented a prototype tool and verified a number of safety properties of SDN controller programs.

The rest of this paper is organized as follows. In Section 2, we review declarative networks and NDLog, and describe our analysis at a high-level. Then, we explain our algorithm for non-recursive programs in Section 3. Next, we extend the algorithm to handle recursive programs in Section 4. The case studies are described in Section 5. We discuss related work in Section 6 and then conclude.

2. Overview

We first review declarative networking and NDLog through examples. Then, we present an overview of our analysis.

2.1 Declarative Networking

Declarative networks are specified using Network Datalog (NDLog), which is a distributed recursive query language used for querying network graphs. Declarative queries are a natural and compact way to implement a variety of routing protocols and (overlay) networks. For example, traditional routing protocols such as path vector and distance-vector protocols can be expressed in a few lines of code [28], and the Chord distributed hash table in 47 lines of code [27]. When compiled and executed, these NDLog programs perform efficiently relative to imperative implementations.

NDLog is based on Datalog [41]. A Datalog program consists of a set of declarative rules. Each rule has the form \( p \leftarrow q_1, q_2, ..., q_n \), which can be read as "\( q_1 \) and \( q_2 \) and ... and \( q_n \) implies \( p \)". Here, \( p \) is the head of the rule, and \( q_1, q_2, ..., q_n \) is a list of literals that constitutes the body of the rule. Literals are either predicates with attributes (which are bound to variables or constants), or Boolean expressions that involve function symbols (including arithmetic) applied to attributes, which we call constraints.

Datalog rules can refer to one another in a mutually recursive fashion. Commas are interpreted as logical conjunctions. The names of predicates, function symbols, and constants begin with a lowercase letter, while variable names begin with an uppercase letter. The following example NDLog program computes full reachability between any pair of nodes. In the runtime, derived predicates are stored as tuples in database tables, so we use predicate and tuple interchangeably for the rest of this paper.

**Reachable:**

- Rule d1: \( \text{reachable}(\text{@}X,Y,C) \leftarrow \text{link}(@X,Y,C) \).
- Rule d2: \( \text{reachable}(\text{@}X,Y,C) \leftarrow \text{link}(@X,Z,C), \text{reachable}(Z,Y,C2), C=C1+C2 \).
- Rule d3: \( \text{reachable}(\text{@}X,Y,C) \leftarrow \text{reachable}(\text{@}X,Z,C), \text{link}(Z,Y,C2), C=C1+C2 \).

The program **Reachable** takes as input \( \text{link}(\text{@}X,Y,C) \) tuples, where each tuple corresponds to a copy of an entry in the neighbor table, and represents an edge from the node itself \( (X) \) to one of its neighbors \((Y)\) of cost \(C\). NDLog supports a location specifier in each predicate, expressed with \( \vec{x} \) symbol followed by an attribute. This attribute is used to denote the source location of each corresponding tuple. For example, \( \text{link} \) tuples are stored based on the value of the \( X \) field. The program **Reachable** derives \( \text{reachable}(\text{@}X,Y,C) \) tuples, where each tuple represents the fact that \( X \) has a path to \( Y \) with cost \( C \). Rule d1 derives \( \text{reachable} \) tuples from direct links. Rule d2 and d3 compute transitive reachability: if there exists a link from \( X \) to \( Z \) with cost \( C1 \), and \( Z \) knows about a path to \( Y \) with cost \( C2 \), then \( X \) can reach \( Y \) with cost \( C1+C2 \). Rule d3 is similar to d2.

As our driving example, we will use the following erroneous program. The following non-recursive set of rules computes one-, two-, and three-hop reachability information within a network.

There is an error in rule r2, where \( \text{onehop} \ X \ Z \ C2\) should be \( \text{onehop} \ Y \ Z \ C2\), thus this program cannot derive three-hop paths.

**ThreeHops (With a deliberate error in r2):**

- Rule r1: \( \text{onehop}(X,Y,C) \leftarrow \text{link}(X,Y,C) \).
- Rule r2: \( \text{twohops}(X,Z,C) \leftarrow \text{link}(X,Z,C1), \text{onehop}(X,Z,C2), C=C1+C2 \).
- Rule r3: \( \text{threehops}(X,Y,C) \leftarrow \text{onehop}(X,Z,C1), \text{twohops}(Z,Y,C2), C=C1+C2 \).
- Rule r4: \( \text{threehops}(X,Y,C) \leftarrow \text{twohops}(X,Z,C1), \text{onehop}(Z,Y,C2), C=C1+C2 \).

2.2 Analysis Overview

The static analysis mainly consists of two processes: a process that summarizes all derivations of predicates in an auxiliary data structure, which we call a derivation pool, and a process that queries properties on the derivation pool. NDLog programs are represented abstractly as dependency graphs. Recursive programs are more complicated than non-recursive programs, so we explain the algorithms for non-recursive programs first, before we discuss extensions to support recursive programs. The dependency graph and the properties to be checked are of the same form for both recursive and non-recursive programs. Next, we formally define the dependency graph and the format of the properties.

**Dependency graph**

A dependency graph has two types of nodes: predicate nodes, denoted \( Np \), and rule nodes, denoted \( Nr \). Each predicate node corresponds to a tuple in the program. A predicate node consists of a unique ID for the node, the name of the predicate and its type, and a tag indicating whether the predicate is on a cycle in the graph. The tag cyc means that the node is on a cycle and ncyc means the opposite. Each rule node corresponds to a rule in the program. A rule node consists of a unique ID, the head of the rule, the body of the rule, which is a list of predicates, and the constraints. The edges, denoted \( E \), are directional. Each edge points either from a rule node to the predicate node which is the head of that rule node, or from a predicate node to a rule node where the predicate is in the rule body.

- **Predicate type** \( \tau \) ::= \text{Pred} \ | \text{bt} \ | \tau \ |
- **Dependency graph** \( G \ ::= (Np, Nr, E) \)
- **Predicate node** \( Np \ ::= (nID, \tau, cyc)(nID, \tau, ncyc) \)
- **Rule node** \( Nr \ ::= (rID, hd, body, c) \)
- **Edge** \( E \ ::= (rID, nID)(nID, rID) \)
- **Rule head** \( hd \ ::= p(\vec{x}) \)
- **Rule body** \( body ::= p_1(x_1),...,p_n(x_n) \)
- **Rule constraints** \( c ::= e_1 \ bop e_2|c_1\land c_2|c_1\lor c_2|\exists x.e \)

To make variable substitutions easier, each predicate takes unique variables as arguments. For instance, the following two NDLog rules are equivalent, but we use \( r1 \) as the normal form.

- Rule r1: \( p(x,y) \leftarrow q(x1), s(y1), x=x1, y=y1 \).
- Rule r2: \( p(x,y) \leftarrow q(x), s(y), x=y \).

The dependency graph for **ThreeHops** is shown in Figure 1, where boxes represent nodes in the graph and arrows represent edges in the graph.

**Properties**

We focus on safety properties, which state that bad things have not happened yet. We use trace-based semantics of NDLog [10, 39]. The advantage of trace-based semantics over fixed point semantics is that the order in which predicates are derived can be clearly specified using traces. Fixed point semantics only care about what is derivable in the end, and are not precise enough
to capture transient faults that appear only in the middle of the execution of network protocols.

To make it possible for automated analysis, we restrict the form of properties to be the following:

\[ \varphi = \forall x_1.p_1(x_1) \land \cdots \land \forall x_n.p_n(x_n) \land c_p(x_1 \cdots x_n) \supset \exists q_1.q_1(y_1) \land \cdots \land \exists q_m.q_m(y_m) \land c_q(x_1 \cdots x_n; y_1 \cdots y_m) \]

The meaning of the property is the following: if all of the predicates \( p_i \) are derivable, and their arguments satisfy constraint \( c_p \), then each of the predicate \( q_i \) must be in one of the derivations of \( p_i \), and the constraint \( c_q \) must be true. We implicitly require \( q_i \) to be derived before \( p_i \). A lot of the correctness properties can be specified using formulas of this form. For instance, we can specify the following three properties of our ThreeHops program:

**Q1:** \( \forall x, y, z, \text{threehops } x \ y \ z \supset \exists x', z', \text{twohops } x \ x' \ z' \)

**Q2:** \( \forall x, y, z, \text{threehops } x \ y \ z \supset \exists x_1, x_2, z_1, z_2, z_3, \text{link } x \ x_1 \ z_1 \land \text{link } x_2 \ y \ z_1 \land \text{link } x_2 \ y \ z_3 \)

**Q3:** \( \exists x, y, z, \text{threehops } x \ y \ z \)

Q1 states that to derive threehops \( x \ y \ z \), it is necessary to derive twohops \( x \ x' \ z' \), for some \( x' \) and \( z' \). Q1 does not hold because there are two ways to derive threehops and one of them does not contain such a twohops tuple as a sub-derivation. Q2 states that to derive a threehops tuple, three links connecting those two nodes are necessary. Q2 should hold. Q3 states that threehops tuple is derivable for some \( x \), \( y \), and \( z \).

### 3. Analyzing Non-recursive Programs

In this section, we first explain how to compute the derivation pool for a non-recursive NDLog program. Then, we show how to check properties. Next, we show how to incorporate network constraints into our property checking algorithm. Finally, we prove the correctness of our algorithm and analyze its time complexity.

#### 3.1 Derivation Pool Construction

For a non-recursive program, its derivation pool maps each predicate to the set of all derivation trees rooted at that predicate. It is formally defined as follows.

**Derivation pool** \( dpool \) is a function that maps each predicate to its derivation pool.

**Entries** \( \Delta \) is a set of pairs of a constraint and a derivation tree, denoted \( \Delta \). At a high-level, \( \Delta \) can be instantiated to be a valid derivation of \( \Delta \) using rules in the program, if \( \Delta \) is derivable. A derivation tree, \( \Delta \), is inductively defined. The base tuples, denoted \((BT, p(x))\), are the leaf nodes. A non-leaf node consists of the unique rule ID of the last rule of the derivation, the conclusion of that rule \((p(x))\), and the list of derivation trees for the body predicates of that rule \((D, List)\). We write \( dpool(p) \) to denote \( dpool((\Delta, D, List)) \), which returns \( \Delta \).

Figure 2 and 3 present the main functions used for constructing a derivation pool from a dependency graph. The top-level function GenDp is defined in Figure 2. This function follows the topological order of the nodes in the dependency graph \( G \). We keep track of a working set \( P \), which is the set of nodes whose derivations can be summarized currently. We also keep track of the set of edges that the function has not traversed yet. The function terminates when all of the edges in the dependency graph have been traversed and the derivations for all of the predicates in the dependency graph are built. In the body of GenDp, we remove one predicate node \( p \) from \( P \), and build all derivations for it. A base tuple’s only possible derivation is one with itself as the leaf node. The constraint associated with this derivation is the trivial true constraint \( \top \) (Line 8). When \( p \) is not a base tuple, derivations for tuples that \( p \)’s derivations depend on have been stored in \( dpool \). The GenDp function constructs derivations for \( p \) given the dependency graph and the current derivation pool (explained later).

After the derivations for a predicate \( p \) are constructed, outgoing edges from \( p \) are removed (Line 13), so predicates that depend on \( p \) can be processed in later iterations. Function RemoveEdges removes outgoing edges from \( p \), and outgoing edges from rule nodes that now do not have incoming edges. This may result in predicates enqueued into \( P \) for the next iteration of processing.

Figure GenDs (Figure 3) takes the dependency graph \( D \) and the derivation pool \( p \), and the property pool \( P \).
as arguments, and returns all derivation pool entries for \( p \). The body of \( \text{GENDS} \) calls \( \text{GENDRULE} \) to construct derivations for each rule that derives \( p \). The function \( \text{GENDRULE} \) makes use of List map and fold operations to construct all possible derivations of \( p \) from a rule of the form \( r : p(x_1, y_1), \ldots, p(x_n, y_n) \). \( \text{dpool} \) has already stored all possible derivations for each \( q \). We need to compute all combinations of the derivations for \( q \)'s. The \( \text{LOOKUP} \) function on line 11 collects the list of derivations for one body tuple and the list map function returns the list of derivations for all body tuples. More precisely, the \( \text{LOOKUP} \) function returns a list of tuples of the form \( (\sigma, c, d) \), where \( d \) is a derivation, \( c \) is the constraint associated with that derivation, and \( \sigma \) is a variable substitution. The domain of \( \sigma \) is \( q \)'s arguments in the rule node, and the range of \( \sigma \) is \( q \)'s arguments in the conclusion of the derivations. We need these substitutions because we alpha-rename the derivations. The constraint in the rule node needs to use the correct variables. Line 12 uses list fold operation to generate all possible derivations. Function \( \text{MERGEDLL} \) and \( \text{MERGED} \) are helper functions to generate the list of derivations. Function \( \text{MERGED} \) is the function that takes as arguments, the list of derivations from \( q_m \) to \( q_{m+1} \) and one derivation for \( q_i \), and prepends the derivation for \( q_i \) to the list of derivations from \( q_m \) up to \( q_i \). Here, the substitutions need to be merged and the resulting constraint is the conjunction of the two constraints. Finally on line 14, function \( \text{COMPLETED} \) generates a well-formed derivation for \( p \) using the rule ID and the list of derivations for \( q \)'s. The constraint associated with this derivation of \( p \) is the conjunction of constraints for the derivation of \( q \) and the constraint in the rule body. The substitutions are applied to the constraint \( c \), because all derivations are alpha-renamed and use fresh variables.

**Example Constraint Pool** A simplified derivation pool for one-hop, twohops, and threehops is shown below. To ease presentation, we rewrite the derivation pool using equality constraints. onehop has only one derivation, using rule \( r1 \). A derivation \( D \) is a tuple consisting of four fields: the name of the last rule in the derivation; the conclusion of the derivation; the constraint associated with this derivation; and the list of derivations of the premises of the last rule. We instantiate the rules with concrete variables. The constraint in \( D \) is true, denoted \( \top \); as there is no constraint in \( r1 \). The predicate \( \text{twohops} \) also has only one derivation, using \( r2 \). The premises of \( r2 \) are link and onehop. Since link is a base tuple, we simply represent its derivation as the tuple itself. The sub-derivation of onehop is the same as in the previous case. The constraint for deriving onehop is the conjunction of three constraints: \( c1 \) is the constraint for deriving onehop, \( c2 \) for the base tuple link, and \( c3 \) the rule constraint of rule \( r2 \). Here \( c2 \) is true, because no constraint is imposed on base tuples.

\[
\text{onehop} \
(\text{D} : (r1, \text{onehop} x1 x2 x3, \{\text{link} x1 x2 x3\}))
\]

\[
c = \top
\]

\[
\text{twohops} \
(D : (r2, \text{twohops} x1 x2 x3 \{\text{link} x1 x2 x3, (r1, \text{onehop} x1 x2 z3, \{\text{link} x1 x2 z3\})\}))
\]

\[
c = \top \land \top \land x1 = y1 + y3
\]

\[
\text{threehops} \
D1 : (r3, \text{threehops} x1 x2 x3, \{(r1, \text{onehop} x1 y2 y3, \{\text{link} x1 y2 y3\})\})
\]

\[
c = \top \land \top \land y2 = u1 + x3 = y2 + u3
\]

\[
D2 : (r4, \text{threehops} x1 x2 x3, \{(r2, \text{twohops} y1 y2 x3, \{\text{link} y1 y2 x3, (r1, \text{onehop} y1 y3 u3, \{\text{link} y1 y3 u3\})\})\})
\]

\[
c = \top \land \top \land y1 + u4 + c5 = x3 = y1 + y3
\]

Tuples threehops has two derivations, one uses \( r3 \), the other uses \( r4 \). Both derivations contain sub-derivations of onehop and twohops. The constraints for deriving threehops include constraint for deriving twohops, onehop, and the rule constraint of \( r3 \) (\( r4 \)).

### 3.2 Property Query

Figure 4 shows the property query algorithm for non-recursive programs. The top-level function \( \text{CKPROP} \) takes the derivation pool and the property as arguments. On line 3, we separate the property into the list of predicates to the left of the implication (\( P \)), the constraint to the left of the implication (\( Q \)), the list of predicates to the right of the implication (\( Q \)), and the constraint to the right of the implication (\( c \)). Next, similar to the derivation pool construction, we construct all possible combinations of the derivations of all the \( q \)'s in \( P \) between lines 5 to 9. We omit the definition of \( \text{MERGEDERIVATION} \), as it is similar to \( \text{MERGEDLL} \). The only difference is that we do not need to alpha-rename the derivations. Next, we check that for each possible derivation of \( q \)'s in \( D \), all of \( q \)'s appear in the derivation, and the constraint \( c \) holds (lines 10 to 14) using function \( \text{CKPROP} \). If for all possible derivations of \( q \)'s, we can always find derivations of \( q \)'s such that the constraint \( \varphi \) holds, \( \psi \) holds (line 14).
function CKPROP(dpool, ϕ)
1. (* P is p1 · · · pn, and Q is q1 · · · qa * )
2. (P, cp, Q, cq) ← ϕ
3. (* Get the list of list of derivations for p1 · · · pn * )
4. L ← LOOKUPREC(dpool, P)
5. (* Combine all possible derivations for p1 · · · pn * )
6. Each entry in D also include substitutions that replace
7. free variables in pi with the variable in the derivation *)
8. D ← MERGE DERIVATION L
9. for each (σ, d, c) in D do
10. if σ = invalid(d, σ) then
11. return invalid(d, σ)
12. end function
13. end function
14. function CKPROP(Dc, cp, d, Q, cq)
15. if Check SAT Dc ∧ cp = (sat, σp) then
16. Σ ← LISTMAP (UNIFY d) Q
17. if nil ∈ Σ then
18. (* Some q is not valid in d * )
19. return invalid(d, σp)
20. else
21. (* Find all possible combinations for q1 · · · qa * )
22. Σq ← LISTMAP (UNIFY d) Q
23. if nil ∈ Σq then
24. (* Some q does not appear in d * )
25. return invalid(d, σp)
26. else
27. (* Find a list of substitutions for q1 · · · qa * )
28. Σq ← LISTMAP (UNIFY d) Q
29. if nil ∈ Σq then
30. (* Some q does not appear in d * )
31. return invalid(d, σp)
32. end function
33. end function
34. function CKPROPDC(cδ, cp, d, Q, cq, β, cs)
35. if Check SAT Dc ∧ cp = (sat, σp) then
36. (* Find all occurrences of δ * )
37. Σd is a list of list of substitutions * )
38. Σd ← LISTMAP (UNIFY d) β
39. (* Σd is a list of substitutions. Each substitution
40. in Σd corresponds to one combination of bks in d * )
41. Σd ← MERGELL Σd
42. (* Given Σd = σ1; · · · ; σδ, cδ = Ln−1 c0σδL * )
43. cδ ← CONJ(Σd, cδ)
44. (* Find all occurrences of δ in d * )
45. Σ ← LISTMAP (UNIFY d) Q
46. if nil ∈ Σ then
47. (* Check network constraints * )
48. if Check SAT Dc ∧ cp ∧ (cδ) = (sat, σc) then
49. (* Network constraints are met * )
50. return invalid(d, σc)
51. else
52. return valid
53. else
54. (* Find all possible combinations for q1 · · · qa * )
55. Σq ← LISTMAP (UNIFY d) Q
56. if nil ∈ Σq then
57. (* Network constraints are met * )
58. return invalid(d, σq)
59. else
60. return valid
61. end function

The function CKPROP checks that in the list of derivations
62. d, with constraints cδ, whether all the predicates in Q appear in
63. d, and cp is true. On Line 18, we first check whether all the pks
64. are derivable and constraint cp is satisfiable. If the conjunction
65. of the derivation constraint cδ and cp is not satisfiable, then
66. the precedent of ϕ is false, so ϕ is trivially true for that derivation. So,
67. we return valid in the else branch (Line 40). If the conjunction
68. is satisfiable, then there are substitutions for variables so that all the
69. pks are derivable and the constraint cp is satisfiable. Next, we need
70. to check whether all qks are derivable. On Line 20, function UNIFY
71. identifies a list of occurrences of qk in the derivation d. That is, for
72. each qk appearing in d, UNIFY returns the list of substitutions:
73. (γ1/x; · · · ; γr/x; nil). For example, if qk appears in ϕ
74. and each γi appearing in d, UNIFY returns the list of substitutions:
75. (γ1/x; · · · ; γr/x; nil). If γi does not appear in d, then UNIFY will return an empty list nil.
76. Therefore, on Line 21, we check whether each γi will appear at least
77. once in d. If it is not the case, then we return invalid with the current
78. derivation and one satisfying substitution that makes pks true for
79. constructing a counterexample. Otherwise, we check whether the
80. constraint cδ can be satisfied. Before doing so, on line 30, we first
81. compute the list of all possible combinations of occurrences of qks.
82. Again, the function MERGELL is similar to MERGELL and we
83. omit the details. Now on line 32, for each possible appearance of
84. qks in d, Σqk is a list of substitutions, each of which, when applied
85. to cp, makes cp use the same variables as those in the derivation.
86. We ask whether the negation of cp together with the derivation
87. constraint and the constraint on the arguments of pks are satisfiable.
88. If this is not satisfiable, then we know that there exists a substitution
89. for variables so that the property ϕ holds. Otherwise, we return
90. the derivation and the satisfying substitution that makes pks and qks
91. derivable, but cp false for counterexample construction.

3.3 Network Constraints
92. Sometimes, the network being analyzed has certain network constraints; for instance, every node in the network has
93. only one outgoing link. Our property query algorithm needs to take into consideration these network constraints. If we ignore these
94. constraints, the counterexample generated by the tool may not be
95. useful as the counterexample could violate the network constraints.
Network constraints that our analysis can handle have similar form as the properties: \(\forall x_1, b_1(x_1) \land \cdots \land \forall x_k, b_k(x_k) \supset c_0(x_1 \cdots x_k)\), where \(b_i\) is a base tuple. Figure 5 shows the algorithm for checking properties on networks with constraints. For clarity, we explain the case with one network constraint. Extending the algorithm to handle multiple constraints is straightforward.

The top-level function \(\text{CKProp}(\text{omitted here})\) is almost the same as \(\text{CKProp}\), except that it takes a network constraint \(\varphi_{\text{net}}\) as an additional argument and uses the function \(\text{CKPropDC}\), which additionally checks network constraints compared to \(\text{CKProp}\).

The function \(\text{CKPropDC}\) takes as additional arguments, a list base tuples \(B\) and the constraint \(c_0\) in the network constraint. In the body of \(\text{CKPropDC}\), we first check whether the constraint on \(p_1, s\) is satisfiable (line 2). If it is not, then this derivation does not violate the property we are checking (line 37). Next, between lines 3 to 10, we find all occurrences of the base tuples in the constraint \(\varphi_{\text{net}}\). We find all possible combinations of substitutions for arguments of these base tuples as they appear in the derivation \(d\). For each occurrence of the base tuples, the constraint \(c_0\) needs to be true, so we compute the conjunction of all the \(c_0\)'s. To give an example, if the constraint is \(\forall b(x) \geq x > 0\), if \(d\) has two occurrences of \(b, b(y)\) and \(b(z)\), then \(c_0 = y \geq 0\) which means that \(d\) does not satisfy the property being checked.

Then, we compute the combination of all possible occurrences of \(q_1\) in derivation \(d\) (line 26) as usual, and find the substitutions that make \(q_1\) appear in \(d\). We compute the conjunction of all \(c_0\)'s (line 28). If the conjunction of \(c_{q_1}, c_{q_2}\), the conjunction of all the \(c_0\)'s found in lines 3–10, and the conjunction of all the \(c_0\)'s is satisfiable, then network constraints are met although \(d\) does not satisfy the property being checked, and we report an error (lines 30–34).

3.4 Analysis of the Algorithms

Correctness. We first prove that our derivation pool construction is correct. Lemma 1 states that an entry for a predicate \(p\) in the derivation pool maps to a valid derivation of \(p\) if the constraints of that derivation are satisfiable; and that if a predicate \(p\) is derivable, then there must be a corresponding entry in the derivation pool.

The function \(\text{DGraph}\) generates a dependency graph for \(\text{prog}\), which can be straightforwardly defined. The semantics of NDLog programs are bottom up, so a set of base tuples \(B\) is needed to start the execution of the program. We write \(\sigma' \supseteq \sigma\) to mean that \(\sigma'\) extends \(\sigma\). \(B\) denotes a set of ground base tuples of \(\text{prog}\). We write \(\text{prog}, B \models d:p(t)\) to mean that \(d\) is a derivation of \(p(t)\) using program \(\text{prog}\) and base tuples \(B\). We write \((c, d', n:p(x)) \in \text{dpool}(p)\) to mean that \(c, d'\) is an entry in the derivation pool \(\text{dpool}\) for the predicate \(p\) and that \(d'\) is a derivation tree with \(p(x)\) as the root.

Lemma 1 (Correctness of derivation pool construction). \(\text{DGraph}(\text{prog}) = G = \text{GENDPOOL}(G) = \text{dpool}\).

1. \(\text{If} \ \text{prog}, B \models d:p(t)\), then \(\exists e, \exists \sigma (e(x), d(x) : p(x)) \in \text{dpool}(p)\).

2. \(\text{If} \ (c(x), d(x) : p(x)) \in \text{dpool}(p)\) and \(\exists e, \exists \sigma (e(x), d(x) : p(x))\), then \(\forall x \in \text{dpool}(p)\).

Using the result of Lemma 1, we prove our property checking algorithm is correct with regard to the formula semantics.

Theorem 2 (Correctness of property query). \(\varphi = \forall x_1, p_1(x_1) \land \cdots \land \forall x_n, p_n(x_n) \land G(x_1 \cdots x_n) \supset G(y_1 \cdots y_m)\).

\[\text{DGRAPH}(\text{prog}) = G = \text{GENDPOOL}(G) = \text{dpool},\]

1. \(\text{CKProp}(\text{dpool}, \varphi) = \text{valid} \iff \forall B, \text{prog}, B \models \varphi.\)

2. \(\text{CKProp}(\text{dpool}, \varphi) = \text{invalid} \iff \exists B \text{s.t.} \text{prog}, B \not\models \varphi.\)

When network constraints are provided, we prove that the property checking algorithm is correct with regard to the network constraints on base tuples.

Theorem 3 (Correctness of property query with constraints). \(\varphi \in \forall x_1, p_1(x_1) \land \cdots \land \forall x_n, p_n(x_n) \land G(x_1 \cdots x_n) \supset G(y_1 \cdots y_m)\).

\(\text{DGRAPH}(\text{prog}) = G = \text{GENDPOOL}(G) = \text{dpool},\)

1. \(\text{CKProp}(\text{dpool}, \varphi, \varphi_{\text{net}}, \varphi) = \text{valid} \iff \forall B, \text{prog}, B \models \varphi.\)

2. \(\text{CKProp}(\text{dpool}, \varphi, \varphi_{\text{net}}, \varphi) = \text{invalid} \iff \exists B \text{s.t.} \text{prog}, B \not\models \varphi \land B \not\models \varphi_{\text{net}}.\)

Time complexity. We give an upper bound on the time complexity of the property query algorithm (Figure 4). Given an NDLog program \(\text{prog}\) with \(R\) rules; each rule contains at most \(W\) body tuples. Also assume \(|Q| = m\) and \(|P| = n\). The time complexity of our algorithm is \(O((\text{ROP}^{m\times W} \times m^{n + W})^n)\). Thus, in practice, \(R\) and \(W\) are usually small. For example, in our case study, \(R\) is bounded by 11 and \(W\) is bounded by 5. In this case, \(R\) and \(W\) can be viewed as constants.

In \(\text{CKPropD}\), we assume \(|Q| = m\), and \(d\) contains at most \(D_0\) instances for each \(q_i\) in \(Q\). Also, assume each query of \(Q\) takes a constant time \(t\). Therefore, the size of \(\Sigma_{d}\) in line 29 is bounded by \((D_0)^m\). The running time of the loop (line 30–line 34) is bounded by \((D_0)^m\). Therefore, the time complexity of \(\text{CKPropD}\) is \(O((D_0)^m)\).

The size of \(\text{CKProp}\) in Figure 4 is dominated by the loop in the algorithm, i.e., line 10–line 13. Suppose each \(p_i\) in \(P\) has at most \(D_0\) instances in \(d\), and \(|P| = n\). Then for \(D_0\) in line 10, we have \(|D| \leq (D_0)^n\). Based on the above discussion, each loop takes \(O((D_0)^m)^n\). Therefore, in the worst case, \(\text{CKProp}\), which takes \((O(D_0)^m)^n\), finishes.

We give more detailed estimates of \(D_0\) and \(D_0\). Assume the input NDLog program \(\text{prog}\) has \(R\) rules, and each rule has at most \(W\) body tuples. Define the height \(H\) of a derivation tree as the number of rules on the longest path from the root predicate to any leaf predicate. Also, let \(D_0(k)\) represent the maximum possible number of derivations for a predicate all of whose derivation trees have height at most \(k\). Notice that \(\text{prog}\) is non-recessive, so we have \(k \leq R\), and \(D_0 = D_0(k)\).

We have the following theorem:

Theorem 4. Given \(k \geq 1\) (a natural number), \(D_0(k) \leq R^{\sum_{d=0}^{k} W^d}\).

Proof. We prove Theorem 4 with mathematical induction. The base case is straightforward. When \(k = 1\), we have \(D_0(1) \leq R\). This is true because \(k = 1\) means that the head predicate can only be derived using one rule with base tuples as bodies. Since there are at most \(R\) rules, the maximum number of derivations for the predicate is \(R\). For the inductive case, assume Theorem 4 holds for \(k = n - 1\), which means \(D_0(n - 1) \leq R^{\sum_{d=0}^{n-1} W^d}\). Now consider a predicate \(p'\) whose derivation trees’ maximum height is \(n\). Given a rule \(r\) that derives \(p'\), each body tuple of \(r\) has all its derivation trees’ height bounded by \(n - 1\). Remember that \(r\) has at most \(W\) body tuples. Thus the number of all possible derivations of \(p'\) using rule \(r\) is bounded by \(D_0(n - 1)^W\). Therefore, there could be at most \(R\) rules that derives \(p'\), we have \(D_0(n) \leq R(D_0(n - 1))^W = R^{\sum_{d=0}^{n} W^d}\).
Based on Theorem 4, \( D_p = O(R^{W_R}) \).

Next, we calculate \( D_q \). Given a \( q_i \) in \( Q \), notice that \( q_i \) must appear in one of all derivations of \( p'_i \)s in \( P \). which means the number of \( q_i \)’s appearance is bounded by the number of nodes that could exist in all the derivations. Each such derivation has height at most \( R \), with each node in the derivation having at most \( W \) children. Therefore, the maximum number of nodes in a derivation is \( W^R \). Since we are \( n \) derivations corresponding to \( p'_i \)s in \( P \), we have \( D_q = nW^R \).

Replace \( D_p \) and \( D_q \) in the time complexity of \( \mathcal{C} \) Prop, we have \( O((D_p)^n(D_q)^m) = O((R^{W_R})^nW^{mW^{Rn}}) \).

### 4. Extension to Recursive Programs

The dependency graph for a recursive program contains cycles. The derivation pool construction algorithm presented in Figure 2 does not work for recursive programs because it relies on the topological order of nodes in the dependency graph. In this section, we show how to augment our data structures and algorithms to handle recursive programs.

#### 4.1 Derivation Pool for Recursive Predicates

When \( p \) is recursively defined, \( dpool \) maps \( p \) to a pair \((c, \Delta)\), where \( \Delta \) has the same meaning as before. The additional constraint \( c \) is an invariant of \( p \); \( c \) is satisfiable if and only if \( p \) is derivable.

```
Constraint pool  dpool ::= \cdots \cdot dpool, (nID, p:\tau) \rightarrow (c, \Delta)
Derivation       D  ::= \cdots \cdot (rec, p(\vec{x}))
Annotation       A ::= \cdots \cdot (A, (nID, p:\tau)) \rightarrow (\vec{x}, c)
```

Derivation trees include a new leaf node \((rec, p(\vec{x}))\), where \( p \) appears on a cycle in the dependency graph. This leaf node is a place holder for the derivation of \( p \). We write \( A \) to denote annotations for recursive predicates, provided by the user. \( A \) maps a predicate \( p \) to a pair \((\vec{x}, c)\), where \( \vec{x} \) is the arguments of \( p \) and \( c \) is the constraint which is satisfiable if and only if \( p \) is derivable.

The structure of the derivation pool construction remains the same. We highlight the changes in Figure 6. The main difference is that now when a cycle is reached, the annotations are used to break the cycle. The working set \( P \), which contains the set of nodes that can be processed next, includes not only predicate nodes that do not have incoming edges, but also includes nodes that depend on only body tuples that have annotations. Consider the following scenario: Rule \( r_1 \) derives \( p \) and has two body tuples \( q_1 \) and \( q_2 \). Let \( q_1 \) be a tuple that is not attached to any node \( \tau \), as \( q_1 \) has been processed and \( q_2 \) has an annotation in \( A \). In this case, we will place \( p \) in the working set. The above mentioned change is encoded in the new \texttt{REMOVEEDGES} function in Figure 6.

The second change is in constructing derivation pool entries for a predicate \( p \). In the non-recursive case, each derivation tree of a predicate \( p \) corresponds to the application of a rule to the list of derivation trees for the body tuples of that rule. In the recursive case, if one of the body tuples, say \( q \), is on a cycle, when we process \( p \), \( q \)'s entries in \( dpool \) have not been constructed. However, the constraint under which \( q \) can be derived is given in the annotation \( A \). In this case, we use \((rec, q(\vec{x}))\) as a place holder for derivations for \( q \) and use the constraint in \( A \) as the constraint for this derivation. The change is reflected in the \texttt{LOOKUp} function for collecting possible derivations of the body predicates (lines 21-23).

Finally, annotations need to be verified. The \texttt{GENDs} function checks the correctness of the annotations after all the predicates have been processed (lines 5-15). For a recursive predicate, the derivation pool maps it to a summary constraint and a list of possible derivations (a pair \((c, \Delta)\)). The requirement of the summary constraint for \( p \) is that it has to be satisfiable if and only if there is at least one derivation for the recursive predicate \( p \). That is, this summary constraint has to be logically equivalent to the disjunction of the constraints associated with all possible derivations of \( p \) in \( \Delta \). We consider two cases for a predicate on a cycle of the dependency graph: (1) there is an annotation for it in \( A \) and (2) there is no annotation. For both cases, we need to collect all the possible constraints for deriving \( p \) from \( \Delta \). Function \texttt{EX_DiSI} computes the disjunction of constraints in \( \Delta \). Each constraint is existentially quantified over the arguments that do not appear in \( p \). For case (1), we need to check that the annotation is logically equivalent to the disjunction of the constraints for all possible derivations of \( p \) (line 10). If this is the case, then the annotated constraint together with \( \Delta \) is returned; otherwise, an error is returned, indicating that the invariant doesn’t

---

Figure 6. Construct derivation pools for recursive programs
hold. For case (2), we return the disjunctive formula returned by $\text{EX\_DISJ}$ (Lines 15). When $p$ is not recursive, only $\Delta$ is returned (line 17).

### 4.2 Property Query

We use the same property query algorithm for non-recursive programs. This obviously has limitations, because the derivations of recursive predicates are not expanded. The imprecision of the analysis comes from the following two sources. The first is that derivations represented as $(\text{rec}, p(\bar{x})); \varphi$ may contain predicates needed by the antecedent of the property (the $q_s$ in $\varphi$). Without expanding these derivations, the algorithm may report that $\varphi$ is violated because $q_s$ cannot be found, even though this is not the case in reality. The second is that network constraints cannot be accurately checked. When we find a suitable derivation $d$ that contains all the $q_s$ such that $c_q$ holds, checking the network constraints on $d$ requires us to expand $(\text{rec}, p(\bar{x}))$ in $d$. The algorithm may report that the property holds, even though the witness it finds does not satisfy the network constraints. Similarly, when the algorithm reports that the property does not hold, the counterexample may not satisfy the network constraints. For the analysis to be precise, we would need annotations for recursive predicates to provide invariants for recursive predicates. Our case studies do not require annotations. Expanding the algorithm to handle recursive predicates precisely remains our future work.

### 4.3 Analysis of the Algorithms

**Correctness.** Similar to the non-recursive case, we prove the correctness of derivation pool construction. We only prove the soundness of the query algorithm. Because derivations of recursive predicates are summarized as $(\text{rec}, p(\bar{x}))$, the correctness of the derivation pool construction needs to consider the unrolling of $(\text{rec}, p(\bar{x}))$.

First, we define a relation $d_{\text{pool}} \vdash d_k, \sigma_k \rightsquigarrow d_{k+1}, \sigma_{k+1}$ (for $k \geq 0$) to mean that a derivation $d_k$ with the substitution $\sigma_k$ can be unrolled using derivations to $d_{k+1}$ and to another derivation $d_{k+1}$ and a new substitution $\sigma_{k+1}$. The rules defining this relation allow the $(\text{rec}, \cdot)$ leaves in the derivation to be gradually expanded, starting from the root and moving up the tree.

We write $d_k$ to denote the derivation after unrolling a derivation $d$ for a sequence of $k$ steps: $d_0, \sigma_0 \rightsquigarrow \ldots, d_1, \sigma_1 \rightsquigarrow \ldots, d_{k-1}, \sigma_{k-1} \rightsquigarrow d_k, \sigma_k$. The $\rightsquigarrow$ rules ensure that each $d_i$ has no $(\text{rec}, \cdot)$ leaves from the root up to the $i - 1$th level (Lemma 4 in Appendix B.1). Step $d_{i+1}, \sigma_{i+1} \rightsquigarrow d_k, \sigma_k$ expands the $(\text{rec}, \cdot)$ leaves at the $i$th level of $d_k$, and thus $d_{i+1}$ has no $(\text{rec}, \cdot)$ leaves from the root up to the $i$th level.

**Rule Base** does not extend the derivation. Rule $\text{WKind}$ weakens the index from $k$ to a larger number $n$ when derivation $d$ does not contain any recursive subderivations of form $(\text{rec}, \cdot)$. Given a derivation $(\text{rec}, p(\bar{x})), d_1, \ldots, d_n; \text{nil}$ with no $(\text{rec}, \cdot)$ leaves from the root up to the $k - 1$th level, and whose subderivations $d_1, \ldots, d_k$ that can be expanded to subderivations $d_1', \ldots, d_k'$, which have no $(\text{rec}, \cdot)$ leaves from the root to the $k - 1$th level, then rule $\text{RREC}$ expands $(\text{rec}, p(\bar{x})), d_1', \ldots, d_n; \text{nil}$ to derivation $(\text{rec}, p(\bar{x})), d_1', \ldots, d_n; \text{nil}$, which has no $(\text{rec}, \cdot)$ leaves from the root up to the the $k$th level. The last rule, $\text{REC}$, is the key rule that expands the derivation of the recursive predicate $p$ at the root in one step. Recursive derivation $(\text{rec}, p(\bar{x}))$ with substitution $\sigma$ is expanded to a derivation $d(p(\bar{x})) \in \Delta_\sigma$, whose constraint $c$ is satisifiable for some substitution $\sigma \cup \sigma'$. We write $d_0, \sigma_0 \rightsquigarrow d_k, \sigma_k$ as shorthand notation for the above sequence of steps.

**Lemma 5** (Correctness of derivation pool construction (recursive)).

\text{DGRAF\_PROP} = \Gamma\text{, and GEN\_DPOOL}(\bar{G}, A) = d_{\text{pool}}$

1. If $\text{PROG} \in \Delta_\sigma$, then either $p$ is not on a cycle in $\Gamma$ and $\exists(c(x), \bar{d}(\bar{x})), p(q(x)), d_{\text{pool}}(p), \exists_{\forall} d_{\text{pool}}(p), \exists_{\forall} d_{\text{pool}}(p)$.

2. $\forall \in N$.

(a) $\text{if } e(x), d(x) ; e(p(x)) \in d_{\text{pool}}(p)$ and $\exists d'_{\text{pool}}(p)$, any $d'_{\text{pool}}(p)$, $d'_{\text{pool}}(p)$.

(b) $\exists \exists_{\forall}(\bar{x}), \exists_{\forall}(\bar{x}), \exists_{\forall}(\bar{x})$.

As we discussed in Section 4.2, we cannot show a general correctness theorem without annotations for recursive predicates.
We can only prove the soundness of the algorithm when there is no network constraint.

**Lemma 6** (Soundness of property query).

\( \varphi = \forall x_1, p_1(x_1) \land \cdots \land \forall x_n, p_n(x_n) \geq \exists y_1, q_1(y_1) \land \cdots \land \exists y_m, q_m(y_m) \land \forall x' \ldots \forall x'_n, y_1 \cdots y_m \)

\[ \text{DGRAPH}(\text{prog}) = G \text{ and } \text{GENDPool}(G,A) = \text{dpool} \text{ and } \text{CKPROP}(\text{dpool}, \varphi) = \text{valid implies } \forall B, \text{ prog, } B \models \varphi. \]

**Time complexity.** The time complexity of the property query algorithm on recursive programs is the same as that of non-recursive programs. More concretely, we also use CKPROP to check properties on recursive programs, so the time complexity remains \( O((D_p)^n(D_q)^m t) \). In addition, the estimate about \( D_p \) and \( D_q \) remains the same as in non-recursive case. Observe that the height of a derivation in the derivation pool is still bounded by \( R \). This is because in the derivation pool construction algorithm (Figure 6), each rule node is processed at most once. Therefore a path in a derivation from the root predicate to any leaf predicate could have at most \( R \) rules. So we directly have \( D_p = n W^R R \). For \( D_p \), Theorem 4 also holds true. The base case is unchanged. For the inductive step, the body tuple now could be a recursive tuple with user’s annotation. However, since we do not expand the derivations for recursive tuples, but collect its annotation as a constraint, the number of derivations for the recursive tuple is effectively one, which satisfies the inductive hypothesis. The rest of the proof is the same. In conclusion, Theorem 4 remains true. Together with the bound on the height of derivations (i.e., \( R \)), this means that \( D_p = O(R W^R) \). And the whole complexity remains unchanged.

## 5. Case Study

We apply our tool to the verification of software-defined networking (SDN) applications. SDN is an emerging networking technique that allows network administrators to program the network through well-defined interfaces (e.g., OpenFlow protocol [34]). SDNs intentionally separate the control plane and the data plane of the network. A centralized controller is introduced to monitor and manage the whole network. The controller provides an abstraction of the network to network administrators, and establishes connections with underlying switches. Recently, declarative programming languages have been used to write SDN controller applications [37].

Like any program, these applications are not guaranteed to be bug-free. We show the effectiveness of our tool in validating and debugging several SDN applications. We demonstrate that the tool can unveil problems in the process of SDN application development, ranging from software bugs, incomplete topological constraints and incorrect property specification. All verifications in our case study are completed within one second.

### 5.1 Verification process

We first provide a high-level description of the verification process. When analyzing a program, the user is expected to provide three types of inputs: (1) formal specification of the property in the form discussed in Section 2; (2) formal specification of initial network constraints (e.g., topological constraints and switch default setup); and (3) formal specification of invariants on recursive tuples.

Our tool takes the above user specifications along with the NDLog program as inputs. It first checks the correctness of the invariants on recursive tuples. After invariants are validated, the tool runs the main algorithm for verification, and outputs either “True” if the property holds, or “False” if the property is not valid. For invalid properties, the tool also generates a concrete counterexample to help the programmer debug the program.

### 5.2 Ethernet Source Learning

The first case study we consider is Ethernet source learning, which allows switches in a network to remember the location of end hosts through incoming packets. More specifically, three kinds of entities are deployed in the network: (1) end hosts (servers or desktops) at the edge of the network that send packets to the network through connected switches, (2) switches that forward a packet if the packet matches a flow entry in the forwarding table, or relay the packet to the controller for further instruction if there is a table miss, and (3) a controller that connects to all switches in the network. The controller learns the position of an end host through packets relayed from a switch, and installs a corresponding flow entry in the switch for future forwarding.

**Encoding** We encode the behaviors of each component in NDLog. Figure 7 presents the NDLog encoding of Ethernet Source Learning (progESL). In a typical scenario, an end host initiates a packet and sends it to the switch that it connects to (rh1). The switch recursively looks up its forwarding table to match against the received packet (rs1, rs2). If a flow entry matches the packet, it is forwarded to the port indicated by the “Action” part of the entry (rs3). Otherwise, the switch wraps the packet in an OpenFlow message, and relays it to the controller for further instruction (rs5). On receiving the OpenFlow message, the controller first extracts the location information of the source address in the packet (the OpenFlow message registers incoming port for each packet), and installs a flow entry matching the source address in the switch (rc1). The controller then instructs the switch to broadcast the mismatched packet to all its neighbors other than the upstream neighbor who sent the packet (rc2). Rules rs5 and rs6 specify the reaction of the switch corresponding to Rules rcl and rc2 respectively — the switch either inserts a flow entry into the forwarding table (rs5) or broadcasts the packet (rs6) as instructed.

**Network constraints** We use the following basic network constraints to limit the topology of the network that runs Ethernet source learning.

\[ \varphi_{\text{net}} \text{ initPacket}(\text{Host, Switch, Src, Dst}) \supset \text{Host } \neq \text{ Switch } \land \text{ Host } = \text{ Src } \land \text{Host } \neq \text{ Dst } \land \text{ Switch } \neq \text{ Dst}. \]

\[ \varphi_{\text{ofcon}} \text{ ofcon}(\text{Controller, Switch}) \supset \text{Controller } \neq \text{ Switch}. \]

\[ \varphi_{\text{swToHst}} \text{ swToHst}(\text{Switch, Host, Port}) \supset \text{Switch } \neq \text{ Host } \land \text{ Switch } \neq \text{ Port } \land \text{ Host } \neq \text{ Port}. \]

\[ \varphi_{\text{swToHst}} \text{ swToHst}(\text{Switch1, Host1, Port1}) \supset \text{(Switch1 = Switch2 } \land \text{ Host1 = Host2) } \land \text{Port1 = Port2).} \]

\[ \text{(Switch1 = Switch2 } \land \text{ Port1 = Port2) } \land \text{Host1 = Host2).} \]

We demand that an end host always initiates packets using its own address as source, and the switch it connects to cannot be the source or the destination (constraints on initPacket). In addition, the controller cannot share addresses with switches (constraints on ofcon), and a switch cannot have a link to itself (constraints on single swToHst). Also, each switch should have only one link connecting the neighbor host, and no two hosts can connect to the same port of a switch (constraints on any two swToHsts).

**Verification results** We verify a number of safety properties that are expected to hold in a network running the Ethernet Source Learning program. Table 1 lists all the properties of the program that we investigate. Here we discuss properties in detail.

Property \( \varphi_{\text{ESL}} \) specifies that whenever an end host receives a packet not destined to it, the switch that it connects to has no matching flow entry for the destination address in the packet. Though this
If an EndHost has received a packet that is not destined for its MAC address, then the switch does not have a routing entry for that EndHost's MAC address.

\[ \varphi_{EVL_2} \]

If EndHost has received a packet destined for it, then the switch has a flow entry for the EndHost.

\[ \varphi_{EVL_3} \]

If the switch has a routing entry for a host with MAC address A, it has received a packet sourced from that host in the past.

\[ \varphi_{EVL_4} \]

<table>
<thead>
<tr>
<th>Property</th>
<th>Property description</th>
<th>Formal Specification</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \varphi_{EVL_1} )</td>
<td>If the switch has a routing entry for a host with MAC address A, it has received a packet sourced from that host in the past.</td>
<td>( \forall Switch, \text{ Mac}, \text{ OutPort}, \text{ Priority}, ) ( \exists \text{ flowEntry}(Switch, \text{ Mac}, \text{ OutPort}, \text{ Priority}) ) ( \land \text{ Mac} = A ) ( \exists \text{ Nei}, \text{ DstMac}, ) ( \exists \text{ packet}(Switch, \text{ Nei}, \text{ Mac}, \text{ DstMac}) )</td>
<td>true</td>
</tr>
<tr>
<td>( \varphi_{EVL_2} )</td>
<td>If an EndHost has received a packet that is not destined for its MAC address, then the switch does not have a routing entry for that EndHost's MAC address.</td>
<td>( \forall EndHost, \text{ Switch}, \text{ SrcMac}, \text{ DstMac}, \text{ InPort}, ) ( \exists \text{ OPort}, \exists \text{ Outport}, \text{ Mac}, \text{ Priority}, ) ( \exists \text{ packet}(EndHost, \text{ Switch}, \text{ SrcMac}, \text{ DstMac}) ) ( \exists \text{ swToHst}(Switch, \text{ EndHost}, \text{ OPort}) ) ( \land \text{ flowEntry}(Switch, \text{ Mac}, \text{ Outport}, \text{ Priority}) ) ( \land \text{ DstMac} \neq \text{ EndHost} ) ( \exists \text{ Mac} \neq \text{ DstMac} )</td>
<td>false</td>
</tr>
<tr>
<td>( \varphi_{EVL_3} )</td>
<td>If EndHost has received a packet destined for it, then the switch has a flow entry for the EndHost.</td>
<td>( \forall EndHost, \text{ Switch}, \text{ SrcMac}, \text{ DstMac}, \text{ OPort}, ) ( \exists \text{ packet}(EndHost, \text{ Switch}, \text{ SrcMac}, \text{ DstMac}) ) ( \exists \text{ swToHst}(Switch, \text{ EndHost}, \text{ OPort}) ) ( \land \text{ DstMac} = \text{ EndHost} ) ( \exists \text{ Switch'}, \text{ Mac}, \text{ Outport}, \text{ Priority}, ) ( \exists \text{ flowEntry}(Switch', \text{ Mac}, \text{ Outport}, \text{ Priority}) ) ( \land \text{ Switch'} = \text{ Switch} \land \text{ Mac} = \text{ DstMac} )</td>
<td>false</td>
</tr>
<tr>
<td>( \varphi_{EVL_4} )</td>
<td>If the switch has a flowEntry for a host with MAC address Mac, then there has been a flow table miss in the past for that particular host</td>
<td>( \forall Switch, \text{ Mac}, \text{ OutPort}, \text{ Priority}, ) ( \exists \text{ flowEntry}(Switch, \text{ Mac}, \text{ OutPort}, \text{ Priority}) ) ( \exists \text{ Switch'}, \text{ SrcMac}, \text{ DstMac}, \text{ InPort}, \text{ Priority}, ) ( \exists \text{ matchingPacket}(Switch', \text{ SrcMac}, \text{ DstMac}, \text{ InPort}, \text{ Priority}) ) ( \land \text{ Switch'} = \text{ Switch} \land \text{ Mac} = \text{ Mac} ) ( \land \text{ InPort} = \text{ OutPort} \land \text{ Priority}' = 0 )</td>
<td>true</td>
</tr>
</tbody>
</table>

Table 1. Safety properties of prog\(_{EVL}\) and verification results

5.3 Firewall

Our second case study is a stateful firewall, which is usually deployed at the edge of a corporate network to filter untrusted packets from the Internet. Compared to a stateless firewall, which makes decision purely based on specific fields of a packet, a stateful firewall allows richer access control depending on flow history. For example, the firewall can allow traffic from an outside end host to reach machines inside the local domain only if the communication was initiated by the internal machines.

Encoding We implement a SDN-based stateful firewall, which can set up filtering policies under the instruction of the controller. The detailed encoding of the program can be found in Figure 11. The firewall enforces the following policy: end-hosts in the corporate domain can send traffic to the outside world but, for security reasons, traffic from an end-host outside the domain can only enter the domain if that end-host has previously received traffic from some end-host within the domain. The network is configured as follows. Two types of hosts are connected to a switch: (i) trusted hosts (within the organization) via port 1; and (ii) untrusted hosts (outside the organization) via port 2. Packets from trusted hosts are always forwarded to untrusted hosts. Packets from untrusted hosts are forwarded to trusted hosts only if the source host has previously received a packet from a trusted host. Key tuples generated at each node executing the program are listed in Table 10. Table 11 summarizes all the rules in the firewall program.
Network constraints  The network constraints for the base tuples in the firewall program are given in Figure 12.

Verification results  We verify a number of properties about the stateful firewall, which are listed below. All the properties are valid.

Property $\varphi_{\text{FW}_1}$ states that for every packet a trusted host receives from an untrusted host via Switch, in the past the switch has received a packet sent from some trusted host (via port 1) to the untrusted host (via port 2). Property $\varphi_{\text{FW}_2}$ states that if the flow table on the switch contains an entry between Src (via untrusted port) and Dst (via trusted port), then in the past the switch has received a packet sent from some Host (via trusted port) to Src. Property $\varphi_{\text{FW}_3}$ states that the trusted controller memory records a connection between Switch and a host, then in the past some trusted source had sent a packet to that host.

5.4 A Weak Firewall  

To further evaluate our tool, we modify the above stateful firewall slightly to construct a “weaker” firewall. The new firewall differs by not requiring packets received from the trusted port to be forwarded to the Internet.

```c
#define TRUSTED_PORT 1
#define UNTRUSTED_PORT 2

/*@Switch*/ Program
* a packet from a trusted host via TRUSTED_PORT
* appeared on switch without a forwarding rule
* we know its from a trusted host since it came via
* TRUSTED_PORT forward packet to untrusted hosts
*/

pktReceived(@Dst, Uport, Src, Tport, Switch):-
    pktIn(@Switch, Src, Tport, Dst),
    Uport == UNTRUSTED_PORT,
    Tport == TRUSTED_PORT.

r1 pktReceived(@Dst, Uport, Src, Tport, Switch):-
    pktIn(@Switch, Src, Tport, Dst),
    connection(@Switch, Controller),
    Tport == TRUSTED_PORT.

pktIn(@Switch, Src, Tport, Dst),
    pktIn(@Switch, Src, PortSrc, Dst),
    perFlowRule(@Switch, Src, PortSrc, Dst).

/*@Switch*/ Program
* a packet from a trusted host appeared on
* the switch Forward according to the rule
* The packet may be from a trusted/untrusted source
*/

pktFromSwitch(@Controller, Switch, Src, Uport, Dst):-
    pktIn(@Switch, Src, Uport, Dst),
    connection(@Switch, Controller),
    Uport == UNTRUSTED_PORT.

perFlowRule(@Switch, Src, Uport, Dst):-
    pktFromSwitch(@Controller, Switch, Src, Uport, Dst),
    trustedControllerMemory(@Controller, Switch, Src),
    Uport == UNTRUSTED_PORT,
    Tport == TRUSTED_PORT.
```

Figure 11. NDLog implementation of progFW

Figure 12. Network constraints for the firewall program

```prolog
\varphi_{\text{FW}_1} \forall\text{Switch, Src, SrcPort, Dst},
\exists\text{PortDst, SrcPortSrc, Switch}
\leftarrow
\text{pktReceived(Dst, PortDst, Src, SrcPortSrc, Switch)}
\land \text{PortDst} = 1 \land \text{PortSrc} = 2
\land \text{Controller, Host, HostPort, pktIn(Switch, Host, HostPort, Src)}
\land \text{HostPort} = 1

\varphi_{\text{FW}_2} \forall\text{Switch, Src, SrcPort, Dst, DstPort,}
\text{perFlowRule(Switch, Src, SrcPort, Dst, DstPort)}
\land \text{SrcPort} = 2 \land \text{DstPort} = 1
\land \text{Host, HostPort, pktIn(Switch, Host, HostPort, Src)}
\land \text{HostPort} = 1

\varphi_{\text{FW}_3} \forall\text{Controller, Switch, Host,}
\text{trustedControllerMemory(Controller, Switch, Host)}
\land \text{Src, SrcPort, pktIn(Switch, Src, SrcPort, Host)}
\land \text{SrcPort} = 1
```

Figure 13. Properties for the stateful firewall

Encoding  We present our NDLog implementation of the program (progWeakFW) in Figure 14. Key tuples generated at each node executing the program are listed in Table 2. The firewall forwards traffic from trusted hosts in the local domain without interference (r1), and also notifies the controller of the destination address in the packet (r2). When the firewall receives a packet from the Internet, it relays the packet to the controller for further decision (r4). If the source address was once registered at the controller, the controller would install a flow entry in the firewall (r5), allowing packets of the same flow to access the internal domain in the future (r3).

Network constraints  The network constraints for this modified version of firewall are shown in Figure 5.4. They are the same as those given in firewall, but with an additional link tuple.

Verification results  We check the property $\varphi_{\text{WeakFW}}$ over the weak firewall:

```prolog
\varphi_{\text{WeakFW}} =
\forall\text{Host, Port, Src, SrcPort, Switch,}
\text{pktReceived(Host, Port, Src, SrcPort, Switch)}
\exists\text{Ctrl, trustedControllerMemory(Ctrl, Switch, Src)}
```

The property specifies that source destinations of all packets reaching internal machines are trusted by the controller. Surprisingly, our tool gives a counterexample for this property (Figure 10), which depicts the scenario that an internal machine H3 sends a packet to another internal machine H4 in the same domain through the firewall F1. Because the controller C1 never registers local machines, the property is violated.

In spite of its simplicity, we find the counterexample interesting, because it can be interpreted in different ways; each corresponds
to a different approach to fixing the problem. The counterexample can be viewed as a revelation of a program bug. The programmer can add a patch to the program and re-verify the property over the updated program. Alternatively, the counterexample could be linked to incomplete specification of network constraints that internal machines should never send internal traffic to the firewall. The fix would then be to insert extra constraints over base tuples of the program. In this case, we can change the property specification, to specify that if a packet is from an external machine, then the source address must be registered at the controller before.

In real deployment, it is up to the programmer to decide which interpretation is most appropriate.

### 5.5 Load Balancing

The third case study is load balancing. When receiving packets to a specific network service (e.g., web page requests), a typical load balancer splits the packets on different network paths to balance traffic load. There are a number of strategies for load balancing, e.g., static configuration or congestion-based adjustment. In our case study, we implement a load balancer which load balances traffic towards a specific destination address, and determines the path of a packet based on the hash value of its source address.

#### Encoding

Figure 16 presents our implementation of load balancer in NDLog (\textit{prog}_{LB}). Key tuples generated at each node executing the program are listed in Table 4. We summarize the program in Table 5.

When the load balancer receives a packet, it first inspects its destination address. If the destination address matches the address that the load balancer is responsible for, the load balancer would generate a hash value of the source destination of the packet (r1). The hash value is used to select the server to which the packet should be routed. The load balancer replaces the original destination address in the packet with the address of the selected server, and forwards the packet to the server (r2). In addition, the load balancer has a default rule that forwards traffic not destined to the designated address without interference (r3).

#### Network constraints

The network constraints are the following:

- \( \varphi_{net1}^{LB} \) \initPacket(v1, v2, v3) \( . \)
  \( v1 \neq v2 \land v2 \neq v3 \land v1 \neq v3 \)

- \( \varphi_{net2}^{LB} \) designated(v4, v5) \( . \)
  \( v4 \neq v5 \)

- \( \varphi_{net3}^{LB} \) designated(v9, v10) \( \land \) designated(v11, v12) \( . \)
  \( v9 = v11 \)
  \( v10 = v12 \)

- \( \varphi_{net4}^{LB} \) serverMapping(v6, v7, v8) \( . \)
  \( v6 \neq v7 \land v7 \neq v8 \land v6 \neq v8 \)

- \( \varphi_{net5}^{LB} \) serverMapping(v13, v14, v15) \( \land \) serverMapping(v16, v17, v18) \( . \)
  \( v13 = v16 \land v14 = v17 \)
  \( v15 = v18 \)

- \( \varphi_{net6}^{LB} \) serverMapping(v13, v14, v15) \( \land \) serverMapping(v16, v17, v18) \( . \)
  \( v13 = v16 \land v15 = v18 \)
  \( v14 = v17 \)

#### Verification result

The property that we verify for load balancing is called flow affinity, that is, if two servers receives packets...
requesting the same service—which means the packets share the same initial destination address—the source addresses of the packets must be different. Formally:

\[ \forall \text{Server1, Server2, Src1, Src2}, \]
\[ \text{recvPacket}(\text{Server1, Src1, ServiceAddr}) \]
\[ \text{recvPacket}(\text{Server2, Src2, ServiceAddr}) \]
\[ \land \text{Server1} \neq \text{Server2} \]
\[ \land \text{Src1} \neq \text{Src2} \]

The property does not hold in the given protocol specification, and a counterexample is given by our tool (Figure 17). In the counterexample, two load balancers responsible for different network service could co-exist in the network, and if a server sends packets to both load-balancers, requesting the same service, it is possible that the packets are routed to different servers.

Similar to the case of the firewall, the programmer can fix the counterexample of the load balancer by patching the program, adding network assumption (e.g., assuming no server is connected to two load-balancers), or changing property specification (e.g., “load-balanced packets that are forwarded out of different ports of the load balancer do not share the same source address”).

### 5.6 Ethernet Address Resolution

The final case study we focus on is the Address Resolution Protocol (ARP) in an Ethernet network. End hosts use ARP to request the destination MAC address corresponding to an IP address that they want to communicate to. Traditionally, the ARP requests are broadcast through the domain. In our case study, we replace the broadcast with a centralized controller that answers ARP requests.

**Encoding** Figure 18 presents an implementation of our NDLog encoding of SDN-based ARP (protARP). Key tuples generated at each node executing the program are listed in Table 7. We summarize the program in Table 8.

**Network constraints** The network constraints are the following:

\[ \varphi_{\text{net}}^{LB} \]
\[ \text{initPacket}(v1, v2, v3) \]
\[ v1 \neq v2 \land v2 \neq v3 \land v1 \neq v3 \]
\[ \varphi_{\text{net}}^{LB} \]
\[ \text{designated}(v4, v5) \]
\[ v4 \neq v5 \]
\[ \varphi_{\text{net}}^{LB} \]
\[ \text{designated}(v9, v10) \land \text{designated}(v11, v12) \]
\[ v9 = v11 \land v10 = v12 \]
\[ \varphi_{\text{net}}^{LB} \]
\[ \text{serverMapping}(v6, v7, v8) \]
\[ v6 \neq v7 \land v7 \neq v8 \land v6 \neq v8 \]
\[ \varphi_{\text{net}}^{LB} \]
\[ \text{serverMapping}(v13, v14, v15) \]
\[ \varphi_{\text{net}}^{LB} \]
\[ \text{serverMapping}(v16, v17, v18) \]
\[ v13 = v16 \land v14 = v17 \land v15 = v18 \]
\[ \varphi_{\text{net}}^{LB} \]
\[ \text{serverMapping}(v13, v14, v15) \]
\[ v13 = v16 \land v14 = v17 \land v15 = v18 \]

**Verification results** We verified two safety properties on the ARP program (Figure 9). All properties are valid.

### 5.7 Discussion

We discuss our experience of using the tool and insights obtained from the case studies.

**Cause of property violation** The counterexamples we discuss above reveal a common pattern: when a predicate in the program has multiple derivations, proving properties over the predicate becomes harder. The situation is even worse when a property involves multiple predicates, each with multiple derivations. The increased complexity of predicate derivations makes it error-prone for human programmers to write correct programs or specify correct properties, and serves as the core cause of property violation. Naturally, the fixes we proposed for counterexamples generally fall into two categories: (1) enriching the property specification to include the missing derivations, or (2) changing the program to remove the uncovered derivations.

**Iterative application development** Another observation is that reasonable network assumptions (e.g., topological constraints) helps prune scenarios that would not appear in actual executions, and generate insightful counterexamples. For example, a counterexample may suggest a topology where a switch has a link to itself. A programmer may start with trivial network assumptions and let the tool guide the exploration of corner cases and gradually add (implicit) network assumptions that are not obvious to the programmer. In fact, our tool enables the programmer to iteratively develop applications. The generated counterexamples could help the programmer understand (1) applicable domain of the program (feedback of missing network constraints); (2) implementation correctness (feedback of bugs in the program); and/or (3) expected behavior of the program (feedback of incorrect property specification). After the programmer fix the problem, she or he can redo the verification repeatedly until the specified property holds.

### 6. Related Work

**Network verification.** In recent years, formal verification has received much attention in the network community. There has been a cloud of prior work on network verification focusing on several different aspects. One aspect is the verification of network configurations, where the proposed solutions detect network configuration errors either 1) through static analysis of the configuration file [2, 16, 17, 36, 48], or 2) by analyzing snapshots of the data plane—reflecting the aggregate impact of all configurations—during system execution [21, 22, 32, 50]. These solutions rely heavily on application-specific network models and property specifications, which limits its adoption in more general scenarios. The second aspect is to leverage proof-based and model-checking techniques to verify the correctness of both the design and implementation of network protocols [15, 18, 24, 46, 47]. Such solutions often demand participation of system administrators during the verification phase, and require domain-specific expertise. The third aspect focuses on security properties, such as origin and route authenticity properties, in secure networking protocols that use cryptographic primitives [5, 6, 10, 13, 51].

Most closely related to ours is the work on verifying network protocol design using declarative networking [10, 46, 47]. The general approach of the prior work share similarities with the one of ours—both model the network behavior using trace semantics, and properties are specified and verified on the trace-based model. However, the proposed solution in this paper enables automated static analysis of safety properties and generates counterexamples for debugging purposes, whereas the prior work relies on manual proofs and therefore can handle a richer set of properties.

**SDN verification.** One special case of network verification is SDN verification [1, 8, 9, 20, 23, 40, 45]. For example, VeriCon [8] defines its own special language for modeling SDN controller and switches [8]. A hoare-logic is developed on this language to prove properties of SDN controllers. The proof obligations are translated to constraints and solved by the SMT solver. NICE is a testing tool for SDN controllers written in Python [9]. NICE combines symbolic execution of the controller programs with state-exploration-based model checking. An alternative approach is to verify network configurations generated by SDN controllers in realtime, instead of
verifying the protocols directly [23, 32]. For instance, Anteater reduced SDN data plane verification into SAT problems so that SAT solvers can solve them effectively in practice [32]. NetKAT is a high-level language designed specifically for programming SDN. Its semantics are based on Kleene algebra. The correctness properties of networks programming using NetKAT are tightly connected to the semantics of Kleene algebra, for instance, reachability, way points and traffic separation.

All of these tools are specially designed to analyze SDN controllers or data planes. Modeling and verifying SDN controllers is one example application of our analysis; our analysis can be applied to analyzing other distributed systems expressible in NDLog. On the other hand, in the current state, we can only check simple safety properties, while VeriCon, NICE, and NetKAT can handle more expressive properties.

Verification of declarative programs. Declarative languages have been proposed to model systems in a variety of domains such as networks, mobile agent planning, and algorithms for graph structures (e.g., Network Datalog (NDLog) [29], MELD [7], Linear Meld [14], Netlog [19], DAHL [31], Dedalus [3]). However, there has been few work on analyzing low-level correctness properties of declarative programs. Notably, Wang et al. [46, 47] developed a proof system for proving correctness properties of networking protocols specified in NDLog, where programs are translated into equivalent first-order logic axioms, that is, all the body tuples are derivable if and only if the head tuple is derivable.

7. Conclusion

We presented an automated approach to analyzing and debugging network protocols using declarative networking. By focusing on a specific class of safety properties, we are able to analyze NDLog programs with few annotations. Our algorithm reduces property checking to constraint solving that can be automatically checked by the SMT solver Z3. We analyzed formal properties of our algorithms and implemented a prototype tool on top of RapidNet, a compilation and execution framework for NDLog. Using our tool, we analyzed a number of real-world SDN network protocols. Our tool can unveil problems ranging from software bugs, incomplete topological constraints, and incorrect property specification. When a given safety property is violated, our tool can provide meaningful counterexamples to help debug the protocol specification.

8. Acknowledgment

We thank the anonymous reviewers for their invaluable comments. This work is supported in part by NSF CNS-1218066, CNS-1117052, CNS-1115706, CNS-0845552, CNS-1453392 and AFOSR Young Investigator Award FA9550-12-1-0327.

References

/* total number of possible servers that the * load balancers can send a packet to */
#define NUM_SERVERS 5

/* Initialize Packets*/
r1 packet(@(LoadBalancer, Client, Server) :-
   initPacket(@(Client, Server, LoadBalancer).

/* Packet appearing on LoadBalancer is to be * sent to its designated server */
r2 hashed(@(LoadBalancer, Client, ServerNum, Server) :-
   packet(@(LoadBalancer, Client, Server),
   designated(@(LoadBalancer, DesignatedDst),
   DesignatedDst == Server),
   Value :: f_hashIp(Client),
   ServerNum :: 1+f_modulo(Value, NumServers),
   NumServers :: NUM_SERVER.

r3 recvPacket(@(Server, Client, ServiceAddr) :-
   hashed(@(LoadBalancer, Client, ServerNum, ServiceAddr),
   serverMapping(@(LoadBalancer, Server, ServerNum).

/* Packet appearing on LoadBalancer is NOT to be * sent to its designated server */
r4 recvPacket(@(Server, Client, Server) :-
   packet(@(LoadBalancer, Client, Server),
   designated(@(LoadBalancer, DesignatedDst),
   Server != DesignatedDst),
   ServiceAddr := Server.

Figure 16. NDLog implementation of prog_{LB}

Figure 17. A counter example for property ϕ_{LB}
/* Controller program */

// Install rules on switch
rc1 flowMod(@Switch, SrcMac, InPort) :-
ofconn(@Controller, Switch),
ofPacket(@Controller, Switch, InPort, SrcMac, DstMac).

// Instruct the switch to send out the unmatching packet
rc2 broadcast(@Switch, InPort, SrcMac, DstMac) :-
ofconn(@Controller, Switch),
ofPacket(@Controller, Switch, InPort, SrcMac, DstMac).

/* Switch program */

// Query the controller when receiving unknown packets
rs1 matchingPacket(@Switch, SrcMac, DstMac, InPort, TopPriority) :-
packet(@Switch, Nei, SrcMac, DstMac),
swToHst(@Switch, Nei, InPort),
maxPriority(@Switch, TopPriority).

// Recursively matching flow entries
rs2 matchingPacket(@Switch, SrcMac, DstMac, InPort, NextPriority) :-
m帙chingPacket(@Switch, SrcMac, DstMac, InPort, Priority),
flowEntry(@Switch, MacAdd, OutPort, Priority),
Priority > 0, DstMac != MacAdd, NextPriority := Priority - 1.

// A hit in flow table, forward the packet accordingly
rs3 packet(@OutNei, Switch, SrcMac, DstMac) :-
marchingPacket(@Switch, SrcMac, DstMac, InPort, Priority),
flowEntry(@Switch, MacAdd, OutPort, Priority),
swToHst(@Switch, OutNei, OutPort),
Priority > 0, DstMac == MacAdd.

// If no flow matches, send the packet to the controller
rs4 ofPacket(@Controller, Switch, InPort, SrcMac, DstMac) :-
ofconn(@Switch, Controller),
marchingPacket(@Switch, SrcMac, DstMac, InPort, Priority),
Priority == 0.

// Insert a flow entry into forwarding table
rs5 flowEntry(@Switch, DstMac, OutPort, Priority) :-
flowMod(@Switch, DstMac, OutPort),
ofconn(@Switch, Controller),
maxPriority(@Switch, TopPriority), Priority := TopPriority + 1.

// Following the controller's instruction, send out the packet as broadcast
rs6 packet(@OutNei, Switch, SrcMac, DstMac) :-
broadcast(@Switch, InPort, SrcMac, DstMac),
swToHst(@Switch, OutNei, OutPort), OutPort != InPort.

/* Host program */

// Packet initialization
rh1 packet(@Switch, Host, SrcMac, DstMac) :-
initPacket(@Host, Switch, SrcMac, DstMac),
hstToSw(@Host, Switch, OutPort).

// Receive a packet
rh2 recvPacket(@Host, SrcMac, DstMac) :-
packet(@Host, Switch, SrcMac, DstMac),
hstToSw(@Host, Switch, InPort).

Figure 7. NDLog implementation of prog_{ESL}
A load balancer receives a packet that a client has sent out. A packet sent by host is destined to the load balancer’s designated server. Destination: H3, Source: H6, Destination: H3. A packet appearing on a load balancer is not to be sent to its designated server. Destination: H4, Source: H6, Destination: H4. A counterexample for property $\varphi_{ESL_2}$. A counterexample for property $\varphi_{ESL_3}$. A counterexample for property $\varphi_{WeakFW}$.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pktReceived(@Dst, DstPort, Src, SrcPort, Switch)</td>
<td>$Dst$ has received a packet via the Switch through port $DstPort$, that was originally send by host $Src$ through port $SrcPort$</td>
</tr>
<tr>
<td>pktIn(@Switch, Src, SrcPort, Dst)</td>
<td>A packet sent by host $Src$ through port $SrcPort$ with target host $Dst$ appeared on the switch</td>
</tr>
<tr>
<td>trustedControllerMemory(@Controller, Switch, Host)</td>
<td>Controller stores a link between $Switch$ an (untrusted) $Host$</td>
</tr>
<tr>
<td>connection(@Switch, Controller)</td>
<td>There is a connection between $Switch$ and $Controller$</td>
</tr>
<tr>
<td>perFlowRule(@Switch, Src, SrcPort, Dst, DstPort)</td>
<td>$Switch$ stores in its memory that untrusted host $Src$ is allowed to send packets to trusted host $Dst$</td>
</tr>
<tr>
<td>pktFromSwitch(@Controller, Switch, Src, SrcPort, Dst)</td>
<td>$Switch$ asks $Controller$ to check if untrusted host $Src$ is allow to send a packet to host $Dst$</td>
</tr>
<tr>
<td>link(@Switch, Dst, PortDst)</td>
<td>$Switch$ is linked to $Dst$ via $PortDst$</td>
</tr>
</tbody>
</table>

Table 2. Tuples for $prog_{WeakFW}$

<table>
<thead>
<tr>
<th>Event</th>
<th>Rule</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize Packets</td>
<td>r1</td>
<td>A load balancer receives a packet that a client has sent out.</td>
</tr>
<tr>
<td>A packet appearing on a load balancer is destined to the load balancer’s designated server</td>
<td>r2</td>
<td>A load balancer has received a packet to be sent to its designated destination. It hashes the source and uses that result modulo the number of servers to get a number corresponding to a server. The load balancer matches the integer obtained by hashing to obtain a server to send the packet to.</td>
</tr>
<tr>
<td>A packet appearing on a load balancer is not to be sent to its designated server</td>
<td>r3</td>
<td>The load balancer forwards the packet directly to the destination as prescribed by the packet.</td>
</tr>
</tbody>
</table>

Table 3. Summary of $prog_{LB}$ encoding
initPacket(}@Client, Server, LoadBalancer) Client sends out a packet to LoadBalancer with intended destination Server.

packet(}@LoadBalancer, Client, Server) LoadBalancer received a packet from Client that has destination Server.

designated(}@LoadBalancer, DesignatedDst) For packets arriving on LoadBalancer with destination address DesignatedDst, LoadBalancer determines it path of a packet based on the hash value of its source address.

hashed(}@LoadBalancer, Client, ServerNum, Server) LoadBalancer had received a packet whose destination address matches the address that it is responsible for. LoadBalancer generates a hash value of the source address of Client to obtain an integer ServerNum. ServerNum is uniquely mapped to Server, to which the packet is to be routed.

serverMapping(}@LoadBalancer, Server, ServerNum) LoadBalancer stores the bijective mappings of each destination server to a unique number, ServerNum.

recvPacket(}@Server, Client, ServiceAddr) Server has received a packet from source Client via LoadBalancer.

Table 4. Tuples for prog_\text{LB}

<table>
<thead>
<tr>
<th>Event</th>
<th>Rule</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize Packets</td>
<td>r1</td>
<td>A load balancer receives a packet that a client has sent out.</td>
</tr>
<tr>
<td>A packet appearing on a load balancer is destined to the load balancer’s designated server</td>
<td>r2</td>
<td>A load balancer has received a packet to be sent to its designated destination. It hashes the source and uses that result modulo the number of servers to get a number corresponding to a server.</td>
</tr>
<tr>
<td></td>
<td>r3</td>
<td>The load balancer matches the integer obtained by hashing to obtain a server to send the packet to.</td>
</tr>
<tr>
<td>Packet appearing on a load balancer is not to be sent to its designated server</td>
<td>r4</td>
<td>The load balancer forwards the packet directly to the destination as prescribed by the packet.</td>
</tr>
</tbody>
</table>

Table 5. Summary of prog_{LB} encoding

<table>
<thead>
<tr>
<th>Property</th>
<th>Property description</th>
<th>Formal Specification</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varphi_{ARP_1}$</td>
<td>If any controller sends an ARP response for IP address $IP_A$, then some end host had sent a broadcast ARP request message for $IP_A$.</td>
<td>$\forall Controller, IP_A, Mac_A, DstIP, DstMac,$ $\exists Qmac, arpRequest(Host, DstIp, DstMac, IP_A, Qmac) \land Qmac = 255$</td>
<td>true</td>
</tr>
<tr>
<td>$\varphi_{ARP_2}$</td>
<td>If any controller has a map between IP address $IP_A$ and MAC address $Mac_A$, then host $A$ has sent a broadcast ARP request.</td>
<td>$\forall Controller, IP_A, Mac_A,$ $\exists Host, SrcIP, SrcMac, DstIP, DstMac,$ $\exists Qmac, arpReply(Host, IP_A, Mac_A, DstIp, DstMac) \land DstMac = 255$</td>
<td>true</td>
</tr>
</tbody>
</table>

Table 6. Results of checking safety properties of prog_{ARP} on our tool
/* constants */
#define BROADCAST "ff:ff:ff:ff:ff"
#define ALL_PORT 0
#define ARP_TYPE "ARP"
#define IPV4_TYPE "IPV4"
#define CONTROLLER "controller"
#define ARP_REQUEST 1
#define ARP_REPLY 2
#define ARP_PRIO 1

/* Host program */

// Send ARP request to directly connected switch
rh1 packet(@Switch, Host, DstMac, DstIp, SrcMac, SrcIp, Arptype) :-
  linkHst(@Host, Switch, Port),
  arpRequest(@Host, SrcIp, SrcMac, DstIp, DstMac),
  Host == SrcIP, Arptype := ARP_REQUEST, DstMac == BROADCAST.

// Received packet from switch and extract ARP reply packets
rh2 arpReply(@Host, SrcIp, SrcMac, DstIp, DstMac) :-
  linkHst(@Host, Switch, Port),
  packet(@Host, Switch, DstMac, DstIp, SrcMac, SrcIp, Arptype),
  Arptype == ARP_REPLY, Type == ARP_TYPE, DstMac == Host.

/* Controller program */

// Register host position
rc1 hostPos(@Controller, SrcIp, Switch, InPort) :-
  ofconnCtl(@Controller, Switch),
  packetIn(@Controller, Switch, InPort, DstMac, DstIp, SrcMac, SrcIp, Arptype),
  Arptype == ARP_REQUEST, DstMac == BROADCAST.

// Recover ARP request
rc2 arpReqCtl(@Controller, SrcIp, SrcMac, DstIp, DstMac) :-
  packetIn(@Controller, Switch, InPort, DstMac, DstIp, SrcMac, SrcIp, Arptype),
  ofconnCtl(@Controller, Switch), Arptype == ARP_REQUEST.

// Learn ARP mapping
rc3 arpMapping(@Controller, SrcIp, SrcMac) :-
  arpReqCtl(@Controller, SrcIp, SrcMac, DstIp, DstMac).

// Generate ARP reply
rc4 arpReplyCtl(@Controller, DstIp, Mac, SrcIp, SrcMac) :-
  arpReqCtl(@Controller, SrcIp, SrcMac, DstIp, DstMac),
  arpMapping(@Controller, DstIp, Mac).

// Send out packet_out message
rc5 packetOut(@Switch, Controller, Port, DstMac, DstIp, SrcMac, SrcIp, Arptype) :-
  arpReplyCtl(@Controller, SrcIp, SrcMac, DstIp, DstMac),
  ofconnCtl(@Controller, Switch),
  hostPos(@Controller, DstIp, Switch, Port), Arptype := ARP_REPLY.

/* Switch program*/

rs1 packetIn(@Controller, Switch, InPort, DstMac, DstIp, SrcMac, SrcIp, Arptype) :-
  ofconnSwc(@Switch, Controller),
  packet(@Switch, Host, DstMac, DstIp, SrcMac, SrcIp, Arptype),
  linkSwc(@Switch, Host, InPort),
  flowEntry(@Switch, Arptype, Prio, Actions),
  Prio == ARP_PRIO, Actions == CONTROLLER, DstMac == BROADCAST.

rs2 packet(@Host, Switch, DstMac, DstIp, SrcMac, SrcIp, Arptype) :-
  packetOut(@Switch, Controller, OutPort, DstMac, DstIp, SrcMac, SrcIp, Arptype),
  linkSwc(@Switch, Host, OutPort), Arptype == ARP_REPLY.

Figure 18. NDLog implementation of progARP
An ARP reply message answering the main request. The controller registers the information that the host with Source IP address has sent a broadcast ARP request message, then some end host receives an ARP reply from the connected switch and stores the message. If any controller sends an ARP response, the controller registers the location of the source address and the MAC address of the host. The controller remembers the information that the host with Source IP address has connected the port InPort of Switch.

### Formal Specification

Receives an ARP request message and relays it to the controller for address resolution.

**Rule**

Switch sends an OpenFlow message to Controller.

**Predicate**

switchIn(Host, Switch, InPort, SrcIp, SrcMac, DstIp, DstMac)

**Description**

Controller has a connection to Switch.

**Property**

rc1

**Description**

Controller remembers that the host of IP address SrcIp has the MAC address SrcMac.

**Property**

rc2

**Description**

An ARP request message at Controller, querying the corresponding MAC address of DstIp, from the host with IP address SrcIp and MAC address SrcMac.

**Predicate**

arpReqCt(Host, SrcIp, SrcMac, DstIp, DstMac)

**Description**

An ARP request message at Controller, querying the corresponding MAC address of DstIp, from the host with IP address SrcIp and MAC address SrcMac.

**Predicate**

arpReplyCt(Host, DstIp, DstMac, SrcIp, SrcMac)

**Description**

An ARP reply message answering SrcMac of SrcIp to the host with IP address DstIp and MAC address DstMac.

**Predicate**

packetOut(Switch, Controller, Port, DstIp, DstMac, SrcIp, SrcMac, Arptype)

**Description**

An OpenFlow message sent from Controller to Switch, to send an ARP packet of type Arptype from SrcIp, SrcMac to DstIp, DstMac.

**Predicate**

flowEntry(Switch, Arptype, Prio, Actions)

**Description**

A flow entry of priority Prio at Switch that applies Actions to packets of type Arptype.

### Table 7. Tuples for $prog_{ARP}$

<table>
<thead>
<tr>
<th>Role</th>
<th>Rule</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>rh1</td>
<td>Host sends an ARP request to a switch that is directly connected to it</td>
</tr>
<tr>
<td>Controller</td>
<td>rc1</td>
<td>Receives an ARP request and registers the location of the source address</td>
</tr>
<tr>
<td></td>
<td>rc2</td>
<td>Receives an ARP request and extracts core information related to address resolution</td>
</tr>
<tr>
<td></td>
<td>rc3</td>
<td>Record the mapping between source IP address and source MAC address</td>
</tr>
<tr>
<td></td>
<td>rc4</td>
<td>Looks up destination IP address of the ARP request in the local ARP cache, and generates an ARP reply packet to answer to request</td>
</tr>
<tr>
<td></td>
<td>rc5</td>
<td>Wraps the ARP reply packet inside an OpenFlow message instructing the switch to relay the ARP reply back to the requesting host</td>
</tr>
<tr>
<td>Switch</td>
<td>rs1</td>
<td>Receives an ARP request message and relays it to the controller for address resolution</td>
</tr>
<tr>
<td></td>
<td>rs2</td>
<td>Follows the controller’s instruction and relays the ARP reply back to the requesting host</td>
</tr>
</tbody>
</table>

### Table 8. Summary of $prog_{ARP}$ encoding

<table>
<thead>
<tr>
<th>Property</th>
<th>Property description</th>
<th>Formal Specification</th>
<th>Result</th>
</tr>
</thead>
</table>
| $\varphi_{ARP_1}$ | If any controller sends an ARP response for IP address $IP_A$, then some end host had sent a broadcast ARP request message for $IP_A$. | $\forall Controller, \forall IP_A, Mac_A, DstIP, DstMac,$  
$\exists Qmac,$  
$\exists \text{arpReplyCt}([Controller, IP_A, Mac_A, DstIP, DstMac]) \supset$  
$\exists \text{arpRequest}([Host, DstIP, DstMac, IP_A, Qmac]) \land Qmac = 255$ | true   |
| $\varphi_{ARP_2}$ | If any controller has a map between IP address $IP_A$ and MAC address $Mac_A$, then host $A$ has sent a broadcast ARP request. | $\forall Controller, \forall IP_A, Mac_A,$  
$\exists Host, SrcIP, SrcMac, DstIP, DstMac,$  
$\exists Qmac,$  
$\exists \text{arpReply}([Host, IP_A, Mac_A, DstIP, DstMac]) \supset$  
$\exists \text{arpMapping}([Controller, IP_A, Mac_A]) \land DstMac = 255$ | true   |

### Table 9. Results of checking safety properties of $prog_{ARP}$ on our tool

20
Table 10. Tuples for $prog_{FW}$

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pktReceived(@Dst, DstPort, Src, SrcPort, Switch)</td>
<td>Host $Dst$ has received a packet via the Switch through port $DstPort$, that was originally send by host $Src$ through port $SrcPort$</td>
</tr>
<tr>
<td>pktIn(@Switch, Src, SrcPort, Dst)</td>
<td>A packet sent by host $Src$ through port $SrcPort$ with target host $Dst$ appeared on the switch</td>
</tr>
<tr>
<td>trustedControllerMemory(@Controller, Switch, Host)</td>
<td>Controller stores a link between $Switch$ an (untrusted) $Host$.</td>
</tr>
<tr>
<td>connection(@Switch, Controller)</td>
<td>There is a connection between $Switch$ and Controller</td>
</tr>
<tr>
<td>perFlowRule(@Switch, Src, SrcPort, Dst, DstPort)</td>
<td>$Switch$ stores in its memory that untrusted host $Src$ is allowed to send packets to trusted host $Dst$</td>
</tr>
<tr>
<td>pktFromSwitch(@Controller, Switch, Src, SrcPort, Dst)</td>
<td>$Switch$ asks $Controller$ to check if untrusted host $Src$ is allow to send a packet to host $Dst$</td>
</tr>
</tbody>
</table>

Table 11. Summary of $prog_{FW}$ encoding

<table>
<thead>
<tr>
<th>Rule</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>A packet from a trusted host, with destination an untrusted host, appeared on switch without a forwarding rule. Forward the packet to the untrusted host.</td>
</tr>
<tr>
<td>r2</td>
<td>A packet from a trusted host appeared on switch without a forwarding rule. Insert the target host $Dst$ of the packet into trusted controller memory.</td>
</tr>
<tr>
<td>r3</td>
<td>A packet from with a forwarding rule appears on the switch, which forwards it according to its flow table</td>
</tr>
<tr>
<td>r4</td>
<td>A packet from an untrusted host appeared on switch, which sends it to the controller to check if it can forward the packet to its intended destination</td>
</tr>
<tr>
<td>r5</td>
<td>Controller checks a packet originally sent by an untrusted host, found that there is a previous link between that untrusted host and the switch, and tells the switch that it can forward the packet by inserting a per flow rule into the switch for that untrusted host</td>
</tr>
</tbody>
</table>
A. Notations, Definitions, and Theorems for Non-Recursive Case

A.1 Notations

We use several notations throughout in the proof.

We write \( \vec{x} \) to denote an ordered list of variables \( x_1, \ldots, x_n \).
\( \vec{y} \subseteq \vec{z} \) means that \( \forall v, v \in \vec{z} \) implies \( v \in \vec{y} \).
\( \vec{z} \subseteq \{ z_1, \ldots, z_n \} \) means that \( \exists i \in \{1, 2, \ldots, n\} \) s.t. \( z_i \in \vec{z} \).
We write \( d(\vec{x}) : \vec{y} \rightarrow \vec{z} \) to denote that \( \exists \vec{e}, \vec{d} \in dpool(p) \) s.t. \( D = d(\vec{e}) \) and \( \vec{e} \). \( \vec{d} \rightarrow \vec{z} \).
where variables \( \vec{y} \) replace variables \( \vec{x} \), or \( \sigma = [\vec{y}/\vec{x}] \), where ground terms \( \vec{y} \) replace variables \( \vec{x} \). We write \( \sigma_1 \sigma_2 \) to abbreviate \( \sigma_1 \circ \sigma_2 \), the successive application of two substitutions. \( \sigma' \triangleright \sigma \) denotes substitution extension, where \( \forall s, s \in \sigma \) implies \( s \in \sigma' \). Substitution projection is denoted by \( \sigma[x] = \sigma' \), meaning that \( \forall v, x \in \vec{x}, \sigma(x) = \sigma'(x) \).
Supposing that \( \vec{y} \) are variables corresponding to \( \vec{x} \), then \( [\sigma[\vec{x}]/\vec{y}] \) means that range\( ([\sigma[\vec{x}]/\vec{y}] = \) range\( ([\sigma[\vec{x}]/\vec{x}] \). \( \bigcup_{i=1}^{n} \sigma_i \) is a union of substitutions \( \sigma_1, \sigma_2, \ldots, \sigma_n \). Finally, \( \bigcup_{i=1}^{n} \sigma_i \) is a union of the disjoint substitutions \( \sigma_1, \sigma_2, \ldots, \sigma_n \), thus \( i \neq j \) implies \( \dom(\sigma_i) \cap \dom(\sigma_j) = \emptyset \). By construction, thus \( \bigcup_{i=1}^{n} \sigma_i \) implies \( \bigcup_{i=1}^{n} \sigma_i \).

A.2 Definitions

There are several definitions that we use throughout in the proofs. We detail them below:

Definition 1 (Bottom up evaluation of a proof). \( \text{prog}, B \vdash d(p) \) if \( d \) is a concrete derivation of predicate \( p \). \( B \) is a collection of base tuples, and \( d \) is generated by bottom-up evaluation of program \( \text{prog} \) initialized with base tuples in \( B \).

As a consequence of Bottom up evaluation of a proof (Definition 1), we write \( d(p) \) to mean that \( \exists \text{prog}, B \vdash d(p) \).

Definition 2 (Predicate appeared in the past).
\( \text{prog}, \exists \sigma \in \text{static} \vdash \exists d \in dpool(p) \) s.t. \( D \in d. \)

Predicate appeared in the past (Definition 2) tells us that \( q(\vec{W}_q) \) has appeared in a past derivation, i.e. there is some derivation of \( p \) containing a subderivation that \( q(\vec{W}_q) \) occurred in.

Definition 3 (Query Property).

\( \varphi = \forall x_1 \vdash p(x_1) \wedge \ldots \wedge \forall x_n \vdash p(x_n) \) and \( c_p(x_1, \ldots, x_n) \).
\( \exists y_1 \vdash q(y_1) \wedge \ldots \wedge \exists y_m \vdash q(y_m) \).

A.3 Correctness of Derivation Pool Construction

Theorem 1 (Correctness of Derivation Pool Construction).

\( \text{DGRAPH}(\text{prog}) = G \) and \( \text{DPOOL}(G) = dpool \)
1. If \( \text{prog}, B \vdash d' : p(\vec{t}) \), then \( \exists \sigma \text{ s.t. } (c(\vec{x}), d(\vec{x}_d) : p(\vec{t})) \in \text{dpool}(p), d(\vec{x}_d)\sigma = d' \text{ and } \vdash c(\vec{x})\sigma. \)

2. If \( (c(\vec{x}), d(\vec{x}_d) : p(\vec{t})) \) \( \in \) \( \) \( \text{dpool}(p) \) \text{ and } \( \vdash c(\vec{x})\sigma \) where \( \text{dom}(\sigma) = \vec{x}_d \), then \( \forall \sigma' \text{ s.t. } \sigma' \geq \sigma \text{ and } \text{dom}(\sigma') = \vec{x}_d, \exists B \text{ s.t. } B = \{ b \mid b \text{ is a base tuple and appears in } d(\vec{x}_d)\sigma' \} \text{ and } \text{prog}, B \vdash d(\vec{x}_d)\sigma' : p(\vec{t})\sigma'. \)

\[
\begin{align*}
\text{Proof.} \\
&\text{1. Proof by induction on the structure of the derivation } d'. \\
\end{align*}
\]

\[
\begin{align*}
\text{Base Case: } d' = (\text{BT, } p(\vec{t})) \\
\text{By assumption,} \\
\text{prog, B} \vdash (\text{BT, } p(\vec{t})): p(\vec{t}) \\
\text{By Function GENDPOOL,} \\
(1) (\top, (\text{BT}, p(\vec{x})) \in \text{dpool}(p)) \\
(2) \text{Define } \sigma = [\vec{t}/\vec{x}] \\
\text{By (1) and (2),} \\
(3) (\text{BT, } p(\vec{x}))\sigma = (\text{BT, } p(\vec{t})) \\
\vdash \top\sigma \\
\text{By (1), (2), and (3),} \\
\text{The conclusion holds} \\
\end{align*}
\]

\[
\begin{align*}
\text{Inductive Case: } d' = (rID, p(\vec{t}), (d'_1: q_1(\vec{t}_1)) : \ldots : (d'_n: q_n(\vec{t}_n)) : \text{nil}) \\
\text{By assumption,} \\
(1) \text{prog, B} \vdash (rID, p(\vec{t}), (d'_1: q_1(\vec{t}_1)) : \ldots : (d'_n: q_n(\vec{t}_n)) : \text{nil}) : p(\vec{t}) \\
\text{By (1),} \\
(2) \forall i \in \{ 1, 2, \ldots, n \}, \text{prog, B} \vdash d'_i : q_i(\vec{t}_i) \\
(3) \text{There is an NDLog rule } rID \text{ which was used in prog to derive } d', \text{ with form:} \\
rID, p(\vec{u}) : - q_1(\vec{u}_1), \ldots, q_n(\vec{u}_n), c_p(\vec{u}, \vec{u}_1, \ldots, \vec{u}_n) \\
\text{By (2), we apply the Induction Hypothesis to obtain:} \\
(4) \forall i \in \{ 1, 2, \ldots, n \}, \\
\exists \sigma_{di} \text{ s.t.} \\
(c_{q_i}(\vec{x}_{di}), d_i(\vec{x}_d)) \in \text{dpool}(q_i), \\
d_i(\vec{x}_d)\sigma_{di} : q_i(\vec{x}_i)\sigma_{di} = d'_i : p(\vec{t}_i), \\
\text{and } \vdash c_{q_i}(\vec{x}_{di})\sigma_{di}. \\
\text{Using (4),} \\
(5) \forall i \in \{ 1, 2, \ldots, n \}, \text{Define } \sigma_{di} = [\vec{t}_d/\vec{x}_d]. \\
\text{By the algorithm,} \\
(6) \forall i \in \{ 1, 2, \ldots, n \}, \vec{x}_i \subseteq \vec{x}_d \text{ and } \vec{x}_{di} \subseteq \vec{x}_d \\
(7) \vec{x}_{d1}, \ldots, \vec{x}_{dn} \text{ are fresh} \\
\text{By GENDPOOL,} \\
(8) (c_p(\vec{z}, \vec{z}_1, \ldots, \vec{z}_n) \wedge \bigwedge_{i=1}^{n} c_{q_i}(\vec{z}_i), \\
rID, p(\vec{z}), (d_1(\vec{z}_1)) : q_1(\vec{z}_1)) : \ldots : (d_n(\vec{z}_n)) : \text{nil}) : p(\vec{z}) \in \text{dpool}(p) \\
\text{where } \vec{z}, \vec{z}_1, \ldots, \vec{z}_n \text{ are fresh} \\
\text{By the semantics of MERGED,} \\
(9) \forall i \in \{ 1, 2, \ldots, n \}, \vec{z}_di \text{ correspond to } \vec{x}_{di} \\
\text{By (6) and (9), we define} \\
(10) \sigma = [\vec{t}/\vec{z}] \cup \bigcup_{i=1}^{n} [\vec{t}_{di}/\vec{z}_{di}] \\
\text{By (8),} \\
\vec{z}, \vec{z}_d, \ldots, \vec{z}_n \text{ are fresh} \\
\sigma \text{ is well-defined} \\
\end{align*}
\]
By (1) and (10),
\[ c_\rho(z_1,\ldots,z_n)\sigma, \]
By (3),
\[ \wedge_{i=1}^n c_\rho(z_i)\sigma \]
Combining the above, we have:
\[ (11) \vdash (c_\rho(z_1,\ldots,z_n) \wedge \wedge_{i=1}^n c_\rho(z_i))\sigma \]
By the definition of \( \sigma \),
\[ (12) \quad (rID, p(\bar{z}), (d_1(z_1^1)\vdash q_1(z_1^1)), \ldots, (d_n(z_n^1)\vdash q_n(z_n^1)))\vdash \nil) \]
\[ = (rID, p(\bar{t}), (d_1^t(q_1(t_1)), \ldots, d_n^t(q_n(t_n)))\vdash \nil) \]
\[ = \bar{d} \sigma \]

By (5), (11) and (12),
The conclusion holds

2. Proof by the structure of derivation \( d \) in \( dpool \)

**Base Case:** \( d = (BT, p(\bar{x})) \)

By assumption,
\[ (1) \quad (\top, (BT, p(\bar{x}))) \in dpool(p) \]
Given any
\[ \sigma' = [\bar{t}/\bar{x}], \]
By (2)
\[ (3) \quad (BT, p(\bar{x}))\sigma' = (BT, p(\bar{t})) \]
(4) Let \( B = \{ p(\bar{t}) \} \)

By (3) and (4),
\[ (5) \quad prog, B \vdash (BT, p(\bar{x}))\sigma' \vdash p(\bar{x})\sigma' \]

**Inductive Case:** \( d = (rID, p(\bar{x}), (d_1(x_1^1)\vdash q_1(x_1^1)), \ldots, (d_n(x_n^1)\vdash q_n(x_n^1)))\vdash \nil) \)

By assumption,
\[ (1) \quad (c_\rho(x_{\bar{y}}), (rID, p(\bar{x}), ((d_1(x_1^1)\vdash q_1(x_1^1)), \ldots, (d_n(x_n^1)\vdash q_n(x_n^1)))\vdash \nil)) \in dpool(p) \]
(2) and \( \vdash c_\rho(x_{\bar{y}})\sigma \)

By \( \text{GENDPOOL}, \)
\( rID \) is the NDLog rule of form \( p(\vec{u}) - q_1(u_1), \ldots, q_n(u_n), c_{pr}(\vec{u}, u_1, \ldots, u_n) \)
(3) \( \forall i \in \{1, 2, \ldots, n\}, \exists c_{pr}(\vec{z}_i), d_i(z_i^1)\vdash q_i(z_i^1) \) \( \in dpool(q_i) \)
(4) \( c_{pr}(x_{\bar{y}}) = c_{pr}(\bar{x}, \bar{z}_1, \ldots, \bar{z}_n) \wedge \bigwedge_{i=1}^n c_{qi}(x_{\bar{y}}) \)
where \( \bar{x}, \bar{z}_1, \ldots, \bar{z}_n \) are fresh variables corresponding to \( z_1, z_2, \ldots, z_n \) respectively

By (2) and (4),
\[ (5) \quad (c_\rho(\bar{x}, \bar{z}_1, \ldots, \bar{z}_n) \wedge \bigwedge_{i=1}^n c_{qi}(x_{\bar{y}}))\sigma = c_{pr}(\bar{t}, t_1^1, \ldots, t_n^1) \wedge \bigwedge_{i=1}^n c_{qi}(t_i^1) \]
and \( \vdash (c_\rho(\bar{x}, \bar{z}_1, \ldots, \bar{z}_n) \wedge \bigwedge_{i=1}^n c_{qi}(x_{\bar{y}}))\sigma \)

By (5),
\[ \forall i \in \{1, 2, \ldots, n\}, \vdash c_{qi}(z_i^1)[t_i^1/z_i^1] \]
Pick any \( \sigma' \) s.t. \( \sigma' \geq \sigma \) and \( \text{dom}(\sigma) = \{ \bar{x}, x_{\bar{y}}', \ldots, x_{\bar{y}}' \} \)
Let \( \sigma_{\bar{y}} = \sigma'_{[x_{\bar{y}}]} \), it is the case that \( \sigma_{\bar{y}} \geq [t_i^1/x_i^1] \)
Let \( \sigma_{\bar{y}} = [t_i^1/x_i^1] \)
By the above, we apply the Induction Hypothesis to obtain:
\[ (7) \quad \exists B_\bar{t} \text{ s.t.} \]
\[ B_\bar{t} = \{ b \mid b \text{ is a base tuple, } b \text{ appears in } d_i(z_i^1)\sigma_{\bar{y}} \} \]
and \( \quad \text{prog, } B_\bar{t} \vdash d_i(z_i^1)\sigma_{\bar{y}} \vdash q_i(z_i^1)\sigma_{\bar{y}} \)
By (7)
\[ (9) \quad d\cdot p(\bar{x})\sigma' = (rID, p(\bar{t}), (d_1(t_1^1)\vdash q_1(t_1^1)), \ldots, (d_n(t_n^1)\vdash q_n(t_n^1)))\vdash \nil) \vdash p(\bar{t}) \]
Define $B = \bigcup_{i=1}^{n} B_i$.

By (7) and (9),
(10) $B = \{ b \mid b$ is a base tuple, $b$ appears in $(rID, p(\bar{x}), (d_1(x_i); q_1(x_i)) ; \cdots ; (d_n(x_n); q_n(x_n)); \text{nil}) \}$
(11) $\text{prog, } B \vdash (rID, p(\bar{x}), (d_1(x_i); q_1(x_i)) ; \cdots ; (d_n(x_n); q_n(x_n)); \text{nil}) \forall \bar{x}$

By (9), (10) and (11),
The conclusion holds \hfill \Box

A.4 Correctness of Property Query

**Theorem 2** (Correctness of Property Query).

$\varphi = \forall \bar{x}_1, p_1(\bar{x}_1) \land \forall \bar{x}_2, p_2(\bar{x}_2) \land \ldots \land \forall \bar{x}_n, p_n(\bar{x}_n) \land c_\bar{p}(\bar{x}_1, \ldots, \bar{x}_n) \supset$

$\exists \bar{y}_1 \land q_1(\bar{y}_1) \land \exists \bar{y}_2 \land q_2(\bar{y}_2) \land \ldots \land \exists \bar{y}_m \land q_m(\bar{y}_m) \land c_q(\bar{x}_1, \ldots, \bar{x}_n, \bar{y}_1, \ldots, \bar{y}_m)$

$\text{DGRAPH(prog)} = \varphi$ and $\text{DPOOL}(\text{prog}) = \text{dpool}$

1. $\text{CKPROP}(\text{dpool}, \varphi) = \text{valid}$ implies $\forall B, \text{prog, } B \vdash \varphi$
2. $\text{CKPROP}(\text{dpool}, \varphi) = \text{invalid}(d, \sigma)$, implies $\exists B$ s.t. $\text{prog, } B \not\vdash \varphi$

**Proof.**

By assumption,
$\text{CKPROP}(\text{dpool}, \varphi) = \text{valid}$

We must show that
1. $\forall B, \text{prog, } B \vdash \varphi$

To show:
2. $\forall B, \forall \sigma_{p_1}, \sigma_{p_2}, \ldots, \sigma_{p_n}$ for the variables $\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n$ respectively,
   and $\forall d_1', d_2', \ldots, d_n'$,

   $\exists \sigma_{q_1}, \sigma_{q_2}, \ldots, \sigma_{q_m}$ for the variables $\bar{y}_1, \bar{y}_2, \ldots, \bar{y}_m$ respectively, s.t.

   Pick any $B$, any substitutions for arguments of $p_1, \ldots, p_n$ of form $\sigma_{p_i} = \{ \bar{t}_i / \bar{x}_i \}$, $\sigma_{p_n} = \{ \bar{t}_n / \bar{x}_n \}$, and any $d_1', \ldots, d_n'$, s.t. (3) $\forall i \in \{1, 2, \ldots, n\}$, $\text{prog, } B \vdash d_i'; p_i(\bar{x}_i) \sigma_{p_i}$

   By Corollary of Derivation Pool (Lemma 1(1)) applied to (3),

   4. $\exists \sigma_{d_i} = \{ \bar{z}_i / \bar{z}_i \}$ and $(c_{p_i} (\bar{z}_i); d_i(\bar{z}_i); p_i(\bar{z}_i)) \in \text{dpool}(p_i)$ s.t.

   5. $\forall i \in \{1, 2, \ldots, n\}$, $\bar{z}_i \subseteq \bar{z}_i$ and $\bar{z}_i \subseteq \bar{z}_i$

   6. $\bar{z}_i, \bar{z}_i, \ldots, \bar{z}_i$ are fresh

By the semantics of LOOKUP and MERGEXERIVATION,

7. $\bar{z}_i, \bar{z}_i$ are variables in $\text{dpool}$ which are arguments for $p_1, \ldots, p_n$, corresponding to $\bar{x}_1, \ldots, \bar{x}_n$

By (7), we can define substitution
8. $\bar{\sigma} = \{ \bar{z}_i / \bar{z}_i \}$

By assumption,

By construction, $\bar{x}_i, \ldots, \bar{x}_n$ are mutually distinct variables

$i \neq j$ implies $\text{dom}([\bar{z}_i / \bar{z}_i]) \cap \text{dom}([\bar{z}_j / \bar{z}_j]) = \emptyset$

Thus $\bar{\sigma}$ is well-defined

Proof of 1.
By (1) and the semantics of \textsc{CkProp},

Every call to \textsc{CkPropD} in the FOR loop on Lines 10-13 of \textsc{CkProp}

By the semantics of \textsc{MergeDerivation},

Every possible combination of derivations for \( p_1, \ldots, p_n \) in \textit{dpool} is found.

There exists one iteration of the FOR Loop on Lines 10-13 of \textsc{CkProp}

where \textsc{CkPropD} is called with the following arguments:

\[ (9) \forall i: \bigwedge_{i=1}^m c_{p_i}(z_{i}^n), \]
\[ c_{p_i}(x_1^i, \ldots, x_n^i) \sigma = c_{p_i}(\bar{z}_1^i, \ldots, z_n^i), \]
\[ d: (d_1(z_1^i):p_1(z_1^i))::(d_2(z_2^i):p_2(z_2^i))::\ldots::(d_m(z_m^i):p_m(z_m^i))::\text{nil} \]
\[ Q: q_1(y_1^i), \ldots, q_m(y_m^i) \text{ are as they appear in } \varphi \]
\[ c_{q_i}(x_1^i, \ldots, x_n^i, y_1^i, \ldots, y_m^i) \sigma = c_{q_i}(\bar{z}_1^i, \ldots, z_n^i, y_1^i, \ldots, y_m^i) \]

The circumstances in which \textsc{CkPropD} returns “valid” are the following:

\textit{Case A}: returns “valid” on Line 35

The true branch of the If-ELSE statement from Lines 18-38 is taken,

The constraints for the derivations of \( p_1, \ldots, p_n \),

and the constraint for \( p_1, \ldots, p_n \) in \( \varphi \) are together satisfiable

The false branch of the If-ELSE statement from Lines 21-35 is taken,

Every \( q_1, \ldots, q_m \) appears in some derivation of \( p_1, \ldots, p_n \),

The false branch of the If-ELSE statement on Lines 32-35 is taken,

The constraints for the derivations of \( p_1, \ldots, p_n \),

and the negation of the constraint for \( q_1, \ldots, q_m \) are together unsatisfiable

\textit{Case B}: returns “valid” on Line 38

The false branch of the If-ELSE statement from Lines 18-38 is taken,

The constraints for the derivations of \( p_1, \ldots, p_n \),

and the constraint for \( p_1, \ldots, p_n \) in \( \varphi \) are together unsatisfiable

We derive a result that will be used in both the cases:

By (4),
\[
c_{p}(x_1^i, \ldots, x_n^i) \sigma_{p_1} \ldots \sigma_{p_n} = c_{p}(x_1^i, \ldots, x_n^i)[t_i/x_i^i] \ldots [t_n^i/x_n^i] = c_{p}(t_1^i, \ldots, t_n^i)
\]
\[
\vdash \equiv c_{p}(\bar{z}_1^i, \ldots, z_n^i) \sigma_{d_1} \ldots \sigma_{d_n}
\]

By (3),
\[
(10) \vdash c_{q}(\bar{z}_1^i, \ldots, z_n^i) \sigma_{d_1} \ldots \sigma_{d_n}
\]

By (10) and (11),
\[
(12) \vdash (\bigwedge_{i=1}^n c_{p_i}(z_{i}^n) \land c_{p}(z_{i}^n)) \sigma_{d_1} \ldots \sigma_{d_n}
\]

\textbf{Part I, Case A:} \textsc{CkPropD} returns “valid” on Line 35

The false branch of the If-ELSE statement from Lines 21-35 is taken, thus

(A.1) \forall j \text{ where } j \in \{1, 2, \ldots, m\},
\[ \exists i \text{ where } i \in \{1, 2, \ldots, n\} \text{ such that } q_j \text{ appears in } d_i(z_i^m) : p_i(z_i^m) \]

The false branch of the If-ELSE statement on Lines 32-35 is taken, thus

(A.2) \forall \sigma_a, \vdash \neg(\bigwedge_{i=1}^n c_{p_i}(z_{i}^n) \land c_{p}(z_{i}^n), \ldots, z_n, \bar{y}_1^i, \ldots, y_m) \sigma_{q_i}) \sigma_a
\]

Thus, we pick \( \sigma_a = \sigma_{d_1} \ldots \sigma_{d_n} \)

(A.3) \vdash \neg(\bigwedge_{i=1}^n c_{p_i}(z_{i}^n) \land c_{p}(z_{i}^n)) \sigma_{d_1} \ldots \sigma_{d_n} \lor (\bigvee_{i=1}^\lambda a_{q_i}) \sigma_{d_1} \ldots \sigma_{d_n}
\]

By (A.3) and (12),
\[ \equiv (\bigvee_{i=1}^\lambda a_{q_i}) \sigma_{d_1} \ldots \sigma_{d_n}, \text{ where } \sigma_{d_\ell} = \bigcup_{j=1}^m [z_{q_j \ell}, y_{j}^\ell / y_{j}^\ell] \]

Thus, there must be some \( \ell \in \{1, 2, \ldots, \lambda\} \) such that

(A.4) \equiv c_{q}(z_{1}, \ldots, z_n, \bar{y}_1^i, \ldots, y_m) \sigma_{d_\ell} \sigma_{d_1} \ldots \sigma_{d_n}
\]

By (A.4), we define

(A.5) \forall j \in \{1, 2, \ldots, m\}, \sigma_{q_j} = [z_{q_j \ell} y_j^\ell / y_j^\ell] \sigma_{d_\ell} = [z_{q_j \ell} y_j^\ell / y_j^\ell] [t_{d_\ell}^\ell / z_{d_\ell}^\ell] = [t_{d_\ell}^\ell / y_j^\ell]
By (4), (A.1) and (A.5),
\[ \forall j \in \{1, 2, \ldots, m\}, (d_1(z_{dn})\sigma_{d1}p_1(x_1)\sigma_{p1})\ldots:(d_n(z_{dn})\sigma_{dn}p_n(x_n)\sigma_{pn})::\text{nil} \vdash \nu q_j(y_j)\sigma_{qj} \]
We can rewrite the above as
(A.6) \[ \forall j \in \{1, 2, \ldots, m\}, (d'_{i1}p_1(x_1)\sigma_{p1})\ldots:(d'_{in}p_n(x_n)\sigma_{pn})::\text{nil} \vdash \nu q_j(y_j)\sigma_{qj} \]
By (3), (4), (A.2), and (A.6),
\[ c_q(x_1, \ldots, x_n, y_1, \ldots, y_m)\sigma_{q1} \sigma_{q2} \ldots \sigma_{qn} \]
By the above,
(A.7) \[ c_q(x_1, \ldots, x_n, y_1, \ldots, y_m)\sigma_{p1} \ldots \sigma_{pn} \sigma_{q1} \ldots \sigma_{qm} \]
By (3), (A.5), (A.6) and (A.7),
the condition specified in (G) holds.

### Part 1, Case B: CKPROP returns “valid” on Line 38

The false branch of the IF-ELSE statement from Lines 18-38 is taken, thus
(B.1) \[ \forall \sigma_p, \vdash -((\bigwedge_{i=1}^n c_{p1}(z_{ni}) \land c_p(z_1, \ldots, z_n))\sigma_p) \]
(12) contradicts (B.1)

## Proof of 2.

By assumption,
\[ \text{CKPROP}(dpool, \varphi) = \text{invalid}(d, \sigma) \]
We must show that show that
\[ \exists B \text{ s.t. prog, } B \not\equiv \varphi \]
To Show:
(1) \[ \exists B, \exists \sigma_{p1}, \sigma_{p2}, \ldots, \sigma_{pn} \text{ for the variables } x_1, x_2, \ldots, x_n \text{ respectively,} \]
and \[ \exists d'_{i1}, d'_{i2}, \ldots, d'_{in} \text{ s.t.} \]
\[ \forall i \in \{1, 2, \ldots, n\}, \text{ prog, } B \vdash d'_{i1}p_i(x_i)\sigma_{p1} \]
and \[ \vdash c_p(x_1, \ldots, x_n)\sigma_{p1} \ldots \sigma_{pn} \]
and \[ \forall q_1, q_2, \ldots, q_m \text{ for the variables } y_1, y_2, \ldots, y_m \text{ respectively,} \]
either \[ \forall j \in \{1, 2, \ldots, m\} \text{ such that } (d'_i p_1(x_i)\sigma_{p1}):(d'_2 p_2(x_2)\sigma_{p2})\ldots:(d'_n p_n(x_n)\sigma_{pn})::\text{nil} \vdash \nu q_j(y_j)\sigma_{qj} \]
or
\[ \forall j \in \{1, 2, \ldots, m\}, (d'_i p_1(x_i)\sigma_{p1}):(d'_2 p_2(x_2)\sigma_{p2})\ldots:(d'_n p_n(x_n)\sigma_{pn})::\text{nil} \not\vdash q_j(y_j)\sigma_{qj} \]
and \[ \vdash -c_q(x_1, \ldots, x_n, y_1, \ldots, y_m)\sigma_{p1} \ldots \sigma_{pn} \sigma_{q1} \ldots \sigma_{qm} \]

By the semantics of the FOR loop on Lines 10-13 of CKPROP,
If a call to CKPROP returns “invalid”, then CKPROP returns “invalid”

By the semantics of LOOKUP and MERGEPRESENTATION,
(2) \[ z_1, \ldots, z_n \text{ are variables in } dpool \text{ which are arguments for } p_1, \ldots, p_n, \text{ corresponding to } x_1, \ldots, x_n \]
Using (2), we can define a substitution
(3) \[ \sigma = \bigcup_{i=1}^n [z_i/x_i] \]

By assumption,
\[ x_1, \ldots, x_n \text{ are mutually distinct variables} \]
\[ i \not= j \text{ implies dom([z_i/x_i])} \cap \text{dom([z_j/x_j])} = \emptyset \]
Thus \[ \sigma \text{ is well-defined} \]

By the semantics of MERGEPRESENTATION, CKPROP is called with arguments of form:
(4) \[ c_q: \bigwedge_{i=1}^n c_{p1}(z_{ni}), \]
and \[ c_{p1}(z_{ni}) \text{ is the constraint for the derivation of } p_1 \text{ in } dpool \]
c_{p}: \[ c_p(x_1, \ldots, x_n) \sigma = c_p(z_1, \ldots, z_n) \]
c_{p}(x_1, \ldots, x_n) is the constraint for \( p_1, \ldots, p_n \) in \( \varphi \)
d: \[ (d_1(z_1)\sigma_1p_1(z_1)):(d_2(z_2)\sigma_2p_2(z_2))\ldots:(d_n(z_{dn})\sigma_n p_n(z_{dn}))::\text{nil} \]
and \[ d_i(z_{di})p_i(z_i) \text{ is the derivation of } p_i \text{ in } dpool \]
Q: \[ q_1(y_1), \ldots, q_m(y_m) \text{ are as they appear in } \varphi \]
c_q: \[ c_q(x_1, \ldots, x_n, y_1, \ldots, y_m) \sigma = c_q(z_1, \ldots, z_n, y_1, \ldots, y_m) \]
and \[ c_q(x_1, \ldots, x_n, y_1, \ldots, y_m) \text{ is the constraint for } q_1, \ldots, q_n \text{ in } \varphi \]
By the algorithm,
(5) \( z_1^1, \ldots, z_n^1 \) are fresh

There are two possible ways for CkPROP\(D \) to return “invalid”

Case A: return “invalid” on Line 23
The true branch of the If-ELSE statement from Lines 18-38 is taken,
The constraints for the derivations of \( p_1, \ldots, p_n \),
and the constraint \( c_\emptyset \) for \( p_1, \ldots, p_n \) in \( \varphi \) are together satisfiable
The true branch of the If-ELSE statement from Lines 21-35 is taken,
Some \( q_i \) does not appear in the derivations of \( p_1, \ldots, p_n \)

Case B: return “invalid” on Line 33
The true branch of the If-ELSE statement from Lines 18-38 is taken,
The constraints for the derivations of \( p_1, \ldots, p_n \),
and the constraint \( c_\emptyset \) for \( p_1, \ldots, p_n \) in \( \varphi \) are together satisfiable
The false branch of the If-ELSE statement from Lines 21-35 is taken,
Every \( q_1, \ldots, q_m \) appears in some derivation of \( p_1, \ldots, p_n \)
The true branch of the If-ELSE statement from Lines 32-35 is taken,
The constraints for the derivations of \( p_1, \ldots, p_n \),
and the constraint for \( p_1, \ldots, p_n \) in \( \varphi \),
and the negation of the constraint \( c_\emptyset \) for \( q_1, \ldots, q_m \) in \( \varphi \) are together satisfiable.

\begin{center}
\textbf{Part 2, Case A: CkPROP\(D \) returns “invalid” on Line 23}
\end{center}

The true branch of the If-ELSE statement from Lines 18-38 is taken, thus
(A.1) \( \exists p_\emptyset \) where \( p_\emptyset = \bigcup_{i=1}^{n} [t_i/z_i^a] \)
\text{s.t.} \( \bigwedge_{i=1}^{n} \exists c_{p_i}(z_i^a) \land c_{p_i}(x_1^i, \ldots, x_n^i) \sigma_p \)
By (6),
\( i \neq j \) implies \( [t_i^c/z_i^a] \cap [t_j^c/z_j^a] = \emptyset \)
Thus \( \sigma_p \) is well-defined

By (A.1),
(A.2) \( \forall i \in \{1, 2, \ldots, n\}, \vdash c_{p_i}(z_i^a) \sigma_p \)
By (4),
(A.3) \( \forall i \in \{1, 2, \ldots, n\}, (c_{p_i}(z_i^a), d_i(z_i^a) \in \text{dpool}(p_i) \)
By the algorithm,
(A.4) \( \forall i \in \{1, 2, \ldots, n\} \), \( z_i^a \subseteq z_i^a \) and \( z_i^a \subseteq z_i^a \).

By Correctness of Derivation Pool Construction (Lemma 1(2)) applied to (A.2) and (A.3),
(A.5) \( \forall i \in \{1, 2, \ldots, n\}, \forall \sigma_{d_i} \text{ s.t. } \sigma_{d_i} \geq \sigma_p|_{z_i^a} \)
\exists B_i \text{ s.t. } B_i = \{ b \mid b \text{ is a base tuple, } b \text{ appears in } d_i(z_i^a) \sigma_{d_i} \}
and \( \text{prog, } B_i \vdash d_i(z_i^a) \sigma_{d_i} : p_i(z_i^a) \sigma_{d_i} \)

By (A.5),
(A.6) we define \( \sigma_{d_i} = \sigma_p|_{z_i^a} \geq \sigma_p|_{z_i^a} \) (let \( \sigma_{d_i} = \{t_i^a/z_i^a\} \)).

Define \( B = \bigcup_{i=1}^{n} B_i \).

By (2) and (A.5), we can define
(A.7) \( \forall i \in \{1, 2, \ldots, n\}, \sigma_{p_i} = \sigma_{d_i} \sigma_{d_i} = \{t_i^a/z_i^a\} \)
By (A.4) and (A.7),
\( \forall i \in \{1, 2, \ldots, n\}, d_i(z_i^a) \sigma_{d_i} : p_i(z_i^a) \sigma_{d_i} = d_i(z_i^a) \sigma_{d_i} : p_i(z_i^a) \sigma_{d_i} \)
By the above,
(A.8) \( \forall i \in \{1, 2, \ldots, n\}, \text{ prog, } B \vdash d_i(z_i^a) \sigma_{d_i} : p_i(z_i^a) \sigma_{d_i} \)

By (A.1),
(A.9) \( \vdash c_{p_i}(x_1^i, \ldots, x_n^i) \sigma_{p_1} \ldots \sigma_{p_n} \)

Since the true branch of the If-ELSE statement from Lines 21-35 is taken, thus
(A.10) \( \exists y_1, \ldots, m \) s.t.
\( q_j \) does not appear in \( d_i(z_i^a) : p_{j_1}(z_i^a) \):\( (d_j(z_i^a) : p_{j_2}(z_i^a)) : \cdots : (d_j(z_i^a) : p_{j_m}(z_i^a)) : \text{nil} \)
By (A.8) and (A.10),
(A.11) \( \forall i = \{1, 2, \ldots, m\} \) s.t.
\( \exists y_1, \ldots, y_m, \)
\( \exists j_1, \ldots, j_m \) s.t.
\( (d_j(z_i^a) : d_{j_1}(z_i^a) : p_{j_1}(z_i^a) : \cdots : (d_j(z_i^a) : p_{j_m}(z_i^a) : \text{nil} \forall q_j(y_j) \sigma_{q_j} \)) \)
By (A.8), (A.9) and (A.11), (1) holds.

The false branch of the IF-ELSE statement from Lines 21-35 is taken, thus
(B.1) $\forall j \in \{1, 2, \ldots, n\}$, 
$\exists z_i$ where $i \in \{1, 2, \ldots, n\}$ such that $q_j$ is in $d_i(z_{di})$.

By the semantics of MERGELL on Line 30, each $\sigma_{qt} \in \Sigma_q$ has form
(B.2) $\sigma_{qt} = \bigcup_{j=1}^{m} [z_{q_j p_{k_j}} / \bar{y}]$
where $1 \leq k_j \leq n$, and $z_{q_j p_{k_j}}$ are the arguments of $q_j$ as they appear in $d_{k_j}(z_{di})$.

By assumption, $y^1, \ldots, y^n$ are mutually distinct variables.
Thus $\sigma_{qt}$ is well-defined.

The true branch of the IF-ELSE statement from Lines 32-35 is taken, thus
(B.3) $\exists \sigma_a = \bigcup_{i=1}^{n} [t_{di} / z_{di}]$ s.t.
$\left( (\wedge_{i=1}^{\lambda} c_p(z_{di}^i)) \wedge \wedge_{i=1}^{\lambda} c_q(z_{di}^i, x_{di}^i, \ldots, \bar{y}) \right) \sigma_a$

By (6), $\sigma_a$ is well-defined.

By (B.3),
(B.4) $\forall i \in \{1, 2, \ldots, n\}, t_{di} \models c_p(z_{di}^i)\sigma_a$

By (4),
(B.5) $\forall i \in \{1, 2, \ldots, n\}, (c_p(z_{di}^i), d_i(z_{di}^i)p_i(z_{di}^i)) \in \text{dpool}(p_i)$

By the algorithm,
(B.6) $\forall i \in \{1, 2, \ldots, n\}, z_{di}^i \subseteq z_{di}$ and $z_{di}^i \subseteq z_{di}$

By Correctness of Derivation Pool Construction (Lemma 1(2)) applied to (B.4) and (B.5), and using (B.6),
(B.7) $\forall i \in \{1, 2, \ldots, n\}, \forall \sigma_{di}, \sigma_{di} \supseteq \sigma_a \models \left[ t_{di} / \bar{z}_{di} \right]$,
$\exists B_i$, s.t. $B_i = \{b \mid b$ is a base tuple, $b$ is in $d_i(z_{di})\sigma_{di}\}$
and $\text{prog}. B_i \models d_i(z_{di})\sigma_{di}$.p_{i}(z_{di})\sigma_{pi}$

By (B.7),
(B.8) we pick $\sigma_{di} = [\sigma_a | z_{di}^i] \geq [\sigma_a | z_{di}^i]$ (let $\sigma_{di} = [t_{di} / \bar{z}_{di}]$) and $B = \bigcup_{i=1}^{n} B_i$.

By (2) and (B.7),
(B.9) $\forall i \in \{1, 2, \ldots, n\}$, let $c_p(z_{di}) / z_{di} = [t_{di} / x_{di}]$

By (B.7) and (B.8),
(B.10) $\forall i \in \{1, 2, \ldots, n\}$, $\text{prog}. B \models d_i(z_{di})\sigma_{di}$.p_{i}(z_{di})\sigma_{pi}$

By the above,
(B.11) $\models c_p(x_1, \ldots, x_n)\sigma_{p_1} \ldots \sigma_{p_n}$

By (B.7) and (B.9),
(B.12) $\forall i \in \{1, 2, \ldots, m\}$,
$\left( (d_1(z_{di})\sigma_{di}$.p_{i}(x_{di})\sigma_{p_{i_1}}) : : (d_2(z_{di})\sigma_{di}$.p_{i}(x_{di})\sigma_{p_{i_2}}) : : \ldots : \ldots : (\ldots : (d_n(z_{di})\sigma_{di}$.p_{i}(x_{di})\sigma_{p_{i_n}}) : : \text{nil} : \neg q_i(\bar{y}))\sigma_{q_i}$

By (B.12),
(B.13) $\exists \sigma_{q_{i_1}} \models \forall \sigma_{q_{i_1}} \models w \leq k_i \leq n$, and $t_{q_{i_1} p_{k_1}} \subseteq \text{range}(\sigma_{d_{k_1}}) = t_{d_{k_1}}$ s.t. $\sigma_{q_{i_1}} = [t_{q_{i_1} p_{k_1}} / \bar{y}]$

This means that $\sigma_{q_{i_1}}$ is a concrete argument for $q_{j_1}$ in a derivation of $p_{k_1}$

By the semantics of UNIFY and MERGELL,
(B.14) $\Sigma_q$ consists of substitutions for $y_1, \ldots, y_m$ (variables of $q_1, \ldots, q_m$, as written in $\varphi$) for all possible occurrences of $q_1, \ldots, q_m$ in $d_{k_1}$.

By (B.13) and (B.14),
(B.15) $\exists \sigma_{qt} \in \Sigma_q$ where $\sigma_{qt} = \bigcup_{i=1}^{m} [z_{q_{j_1} p_{k}} / \bar{y}]$ s.t.
$\forall i \in \{1, 2, \ldots, m\}$,
$q_i(\bar{y})\sigma_{qt}$ is well-defined.
Proof.

Pick any $\forall \sigma$. Goal:

By Correctness of derivation pool construction (Lemma 1(1)),

If there is no such $\sigma$,

By assumption,

By (B.3) and (B.15),

By (2), (B.3), (B.9), and (B.15),

By (B.8), (B.9), (B.10), (B.11), (B.12) and (B.16),

(1) holds

A.5 Correctness Of Property Query With Constraints

Theorem 3 (Correctness Of Property Query With Constraints),

$\varphi \equiv \forall x_1, p_1(x_1) \lor \forall x_2, p_2(x_2) \land \ldots \land \forall x_n, p_n(x_n) \land \exists y_1, q_1(y_1) \land \exists y_2, q_2(y_2) \land \ldots \land \exists y_m, q_m(y_m) \land c_q(x_1, \ldots, x_n, y_1, y_2, \ldots, y_m) = 0$

$\varphi_{net} = \forall u_1, b_1(u_1) \land \ldots \land \forall u_k, b_k(u_k) \supset c_u(u_1, \ldots, u_k)$

DGRAPH($\text{prog}$) = $\mathcal{G}$ and GENDPOL($\text{prog}$) = $dpool$.

(1) $\text{CKPROP}(dpool, \varphi_{net}, \varphi) = \text{valid}$ implies $\forall B$, either $\text{prog}, B \models \varphi$ or $B \not\models \varphi_{net}$.

(2) $\text{CKPROP}(dpool, \varphi_{net}, \varphi) = \text{invalid}(d, \sigma)$ implies $\exists B$ s.t. $\text{prog}, B \not\models \varphi$ and $B \not\models \varphi_{net}$.

Proof.

By assumption,

(1) $\text{CKPROP}(dpool, \varphi_{net}, \varphi) = \text{valid}$

We must show

$\forall B$, either $\text{prog}, B \models \varphi$ or $B \not\models \varphi_{net}$

Goal:

$\forall B,$

either

(2) $\forall \sigma_{p_1}, \sigma_{p_2}, \ldots, \sigma_{p_n}$ for the variables $x_1, x_2, \ldots, x_n$ respectively, and

$\forall d_1^i, d_2^i, \ldots, d_n^i$

$\forall i \in \{1, 2, \ldots, n\}, \text{prog}, B \models d_i^i; p_i(x_i) \sigma_{p_i}$

and $c_p(x_1, \ldots, x_n) \sigma_{p_1}, \ldots, \sigma_{p_n}$ implies

$\exists \sigma_{q_1}, \ldots, \sigma_{q_m}$ for the variables $y_1, y_2, \ldots, y_m$ respectively such that

$\forall j \in \{1, 2, \ldots, m\}, (d_1^j; p_1(x_1) \sigma_{p_1}); (d_2^j; p_2(x_2) \sigma_{p_2}); \ldots; (d_n^j; p_n(x_n) \sigma_{p_n}); \text{nil} \models q_j(y_j) \sigma_{q_j}$

and $c_q(x_1, \ldots, x_n, y_1, y_2, \ldots, y_m) \sigma_{p_1}, \ldots, \sigma_{p_n} \sigma_{q_1}, \ldots, \sigma_{q_m}$

or

(3) $\exists \sigma_{b_1}, \sigma_{b_2}, \ldots, \sigma_{b_k}$ for the variables $u_1^i, u_2^i, \ldots, u_k^i$ respectively, s.t.

$\forall i \in \{1, 2, \ldots, k\}, b_i(u_i) \sigma_{b_i} \in B$

and $\models \neg c_b(u_1^i, u_2^i, \ldots, u_k^i) \sigma_{b_1}, \sigma_{b_2}, \ldots, \sigma_{b_k}$

Pick any $B$, any $\sigma_{p_1}, \ldots, \sigma_{p_n}$ of form $[x_i/x_1], \ldots, [x_n/x_n]$, and any $d_i^j; p_i(x_i) \sigma_{p_i}, d_n^j; p_n(x_n) \sigma_{p_n}$ s.t.

(4) $\forall i \in \{1, 2, \ldots, n\}, \text{prog}, B \models d_i^j; p_i(x_i) \sigma_{p_i}$ and $c_p(x_1, \ldots, x_n) \sigma_{p_1}, \ldots, \sigma_{p_n}$

If there is no such $\sigma_{p_i}$ or $d_i^j$, then $\text{prog}, B \models \varphi$ trivially holds.

By Correctness of derivation pool construction (Lemma 1(1)),

(5) $\forall i \in \{1, 2, \ldots, n\}$.
Let $\exists \sigma_s, \exists \sigma_d, \exists \sigma_{di}$ where $\sigma_{di} = [t_{\bar{d}}/z_{\bar{d}}]$ s.t.
\[ d_1(z_{\bar{d}}) \sigma_d : p_i(z_i) \sigma_d = d_1'(z_i) \sigma_{pi}, \] and $\in c_p(z_i) \sigma_{di}$

By the algorithm,
(6) $q_i \in \{1, 2, \ldots, n\}$, $z_i \subseteq z_{\bar{d}}$ and $z_{\bar{d}} \subseteq z_{\bar{d}}$.
(7) $\bar{z}_{\bar{d}}, \ldots, \bar{z}_{\bar{d}}$ are fresh.

By the semantics of LOOKUP and MERGEPIDERATION,
$\bar{z}_1, \ldots, \bar{z}_n$ are variables in $\text{dpool}$ which are arguments for $p_1, \ldots, p_n$, corresponding to $\bar{x}_1, \ldots, \bar{x}_n$.

By the above, we can define substitution

Thus $\sigma$ is well-defined.

By (1) and the semantics of CKPROP,
---
Every call to CKPROPDC in the FOR loop on Lines 12-15 of CKPROP returned “valid.”
---

By the semantics of MERGEPIDERATION,
---
Every possible combination of derivations for $p_1, \ldots, p_n$ in $\text{dpool}$ is found.
---

Therefore, CKPROPDC is called with the following arguments and returns valid.

(9) $c_p: \bigwedge_{i=1}^{n} c_p(z_{i})$ and $c_p(z_{i})$ is the constraint for the derivation of $p_i$ in $\text{dpool}$
\[ c_p: c_p(x_1, \ldots, x_n) \sigma = c_p(z_1, \ldots, z_n). \]
and $c_p(x_1, \ldots, x_n)$ is the constraint for $p_1, \ldots, p_n$ in $\varphi$

(10) $d_1(z_{\bar{d}}) : p_i(z_i) :: d_1'(z_i)$
\[ (\text{dpool}) \]

By the above and (4),

(10) $\in c_p(z_1, \ldots, z_n) \sigma_{d1} \ldots \sigma_{dn}$

By (5),

(11) $\in (\bigwedge_{i=1}^{n} c_p(z_{i})) \sigma_{d1} \ldots \sigma_{dn}$

By (10) and (11),

(12) $\in (\bigwedge_{i=1}^{n} c_p(z_{i}) \wedge c_p(z_1, \ldots, z_n)) \sigma_{d1} \ldots \sigma_{dn}$

---


---

By the semantics of MERGELL on Line 8, each $\sigma_{b_j} \in \Sigma_b$ has form
(1.1) $\sigma_{b_j} = \bigwedge_{i=1}^{\mu} [z_{\bar{d}i}/\bar{u}_i]$ where $1 \leq j \leq \Sigma_b$.
and $z_{\bar{d}i}$ are the arguments of $b_j$ as they appear in $d_\ell(z_{\bar{d}i}) : p_i(z_i)$ where $\ell \in [1, n]$.

The false branch of the IF-ELSE statement from Lines 15-19 is taken, thus
\[ \forall \sigma_{\ell} \in (\bigwedge_{i=1}^{\mu} c_p(z_i) \wedge \bigwedge_{i=1}^{\mu} c_p(z_i) \wedge \bigwedge_{i=1}^{\mu} c_p(u_1, \ldots, u_k) \sigma_{bj}) \sigma_{\ell} \]

Let $\sigma_{\ell} = \sigma_{d1} \ldots \sigma_{dn}$, then
(2.1) $\in (\bigwedge_{i=1}^{\mu} c_p(z_i) \wedge \bigwedge_{i=1}^{\mu} c_p(z_i)) \sigma_{d1} \ldots \sigma_{dn} \wedge (\bigvee_{i=1}^{\mu} c_p(u_1, \ldots, u_k) \sigma_{be}) \sigma_{d1} \ldots \sigma_{dn}$

By (12) and (2.1),
\[ (\bigvee_{i=1}^{\mu} \neg c_p(u_1, \ldots, u_k) \sigma_{d1}) \sigma_{d1} \ldots \sigma_{dn} \]

There must be some $1 \leq \ell \leq \mu$ s.t.
(2.3) $\in \neg c_b(u_1, \ldots, u_k) \sigma_{d1} \sigma_{d2} \ldots \sigma_{dn}$

We define
(4.1) $\forall i \in \{1, 2, \ldots, k\}, \sigma_{b_i} = ([\sigma_{d1} \ldots \sigma_{dn}) (z_{\bar{d}i})/\bar{u}_i]\]

Thus, by (1.1) and (4.1),
\[ \forall i \in \{1, 2, \ldots, k\}, \exists \sigma_{b_i} (\bar{u}_i) \sigma_{d1} \ldots \sigma_{dn} \]

\[ \exists \sigma_{b_i} (\bar{u}_i) \sigma_{d1} \ldots \sigma_{dn} \]

Thus, by (1.1) and (4.1),
\[ \forall i \in \{1, 2, \ldots, k\}, \exists \sigma_{b_i} (\bar{u}_i) \sigma_{d1} \ldots \sigma_{dn} \]

\[ \exists \sigma_{b_i} (\bar{u}_i) \sigma_{d1} \ldots \sigma_{dn} \]
By (4) and (5) and the semantics of \(prog\), \(B \models d'_i : p(i)\) any base tuple in \(d_z(z_i)\), which equals to \(d'_i\), is also in \(B\).

By the above
\[(A.5)\ b_i(u'_i) \sigma_{b_i} \in B\]

By (A.4) and (A.5),
\[c_b(u'_i, \ldots, u'_b) \sigma_{b_b} \ldots \sigma_{b_d}\]
\[= c_b(u'_i, \ldots, u'_b) \sigma_{b_i} \ldots \sigma_{b_k}\]

Thus,
\[(A.6) \models c_b(u'_1, \ldots, u'_b) \sigma_{b_1} \ldots \sigma_{b_k}\]

By (4), (A.4), (A.5), and (A.6), \(B \not\models \varphi_{net}\).

---

**Part I, Case B: CKPROPDC returns “valid” on Line 34**

The false branch of the IF-ELSE statement on Lines 32-36 is taken, thus
\[
\forall \sigma_s, \models \neg (\bigwedge_{i=1}^{n_c} c_p(z^i_n) \land c_p(z^i_n, \ldots, z^i_0) \land \bigwedge_{i=1}^{l} \neg c_b(\bar{u}_{i-1}, \ldots, \bar{u}_0) \sigma_{b_i}) \sigma_s
\]

Which can be rewritten as
\[
\forall \sigma_s, \models \neg (\bigwedge_{i=1}^{n_c} c_p(z^i_n) \land c_p(z^i_n, \ldots, z^i_0)) \lor (\bigwedge_{i=1}^{l} \neg c_b(\bar{u}_{i-1}, \ldots, \bar{u}_0) \sigma_{b_i}) \sigma_s
\]

Thus, letting \(\sigma_s = \sigma_{d_1} \ldots \sigma_{d_n}\), we have
\[(B.3) \models (\bigwedge_{i=1}^{n_c} c_p(z^i_n) \land c_p(z^i_n, \ldots, z^i_0)) \sigma_s \lor (\bigwedge_{i=1}^{l} \neg c_b(\bar{u}_{i-1}, \ldots, \bar{u}_0) \sigma_{b_i}) \sigma_s
\]

By (12) and (B.3),
\[
\models (\bigwedge_{i=1}^{n_c} c_p(z^i_n) \land c_p(z^i_n, \ldots, z^i_0)) \sigma_s \lor (\bigwedge_{i=1}^{l} \neg c_b(\bar{u}_{i-1}, \ldots, \bar{u}_0) \sigma_{b_i}) \sigma_s
\]

We have two subcases, one is
(I) \[\models (\bigwedge_{i=1}^{n_c} c_p(z^i_n) \land c_p(z^i_n, \ldots, z^i_0)) \sigma_s\]
the other is
(II) \[\models (\bigwedge_{i=1}^{l} \neg c_b(\bar{u}_{i-1}, \ldots, \bar{u}_0) \sigma_{b_i}) \sigma_s\]

For case (I), we can show \(prog, B \models \varphi\) by using the same arguments as the proof of Lemma 2, 1.A.
For case (II), we can show \(B \not\models \varphi_{net}\) by using the same arguments as the proof of the previous case.

---

**Part I, Case C: CKPROPDC returns “valid” on Line 42**

We can show a contradiction.
The proof is the same as the arguments in the proof of Lemma 2, 1.B.

---

**Proof of 2.**

By assumption,
\[\text{CKPROPDC}(d\text{pool}, \varphi_{net}, \varphi) = \text{invalid}(d, \sigma)\]
We must show
(1) \(\exists B\ s.t. prog, B \not\models \varphi\) and \(B \models \varphi_{net}\).

**Goal:**
\[
\exists B\ s.t. \text{the following holds:}
\begin{align*}
(2) &\exists \sigma_{p_1}, \sigma_{p_2}, \ldots, \sigma_{p_n} \text{ for the variables } x_1, x_2, \ldots, x_n \text{ respectively,} \\
&\forall i \in \{1, 2, \ldots, n\}, \text{prog, } B \models d'_i : p_i(x_i) \sigma_{p_i} \\
&\text{and } \exists d'_1, d'_2, \ldots, d'_n, \text{s.t.} \\
&\forall i \in \{1, 2, \ldots, n\}, \text{prog, } B \models d'_i : p_i(x_i) \sigma_{p_i} \\
&\text{and } \exists \sigma_{q_1}, \sigma_{q_2}, \ldots, \sigma_{q_m} \text{ for the variables } y_1, y_2, \ldots, y_m \text{ respectively.} \\
&\text{either } \forall j \in \{1, 2, \ldots, m\} \ s.t. \\
&(d'_i : p_1(x'_i) \sigma_{p_1} : (d'_2 : p_2(x'_2) \sigma_{p_2}) : \ldots : (d'_n : p_n(x'_n) \sigma_{p_n}) \not\models q_j(y'_j) \sigma_{q_j}) \\
&\text{or} \\
&\forall j \in \{1, 2, \ldots, m\}, \\
&(d'_i : p_1(x'_i) \sigma_{p_1} : (d'_2 : p_2(x'_2) \sigma_{p_2}) : \ldots : (d'_n : p_n(x'_n) \sigma_{p_n}) \models q_j(y'_j) \sigma_{q_j}) \\
&\text{and } \exists \sigma_{c_1}, \sigma_{c_2}, \ldots, \sigma_{c_k} \text{ for the variables } u_1, u_2, \ldots, u_k \text{ respectively.} \\
&\forall i \in \{1, 2, \ldots, k\}, \text{b_i(u_i)} \sigma_{b_i} \in B \text{ implies } \models c_b(u'_1, u'_2, \ldots, u'_b) \sigma_{b_1} \sigma_{b_2} \ldots \sigma_{b_k}
\end{align*}
\]

By semantics of the FOR loop on Lines 12-15 of CKPROPDC,
If a call to CKPROPDC returns “invalid” on an iteration, CKPROPDC returns “invalid”
By the semantics of \textsc{LookUp} and \textsc{MergeDerivation},

(4) $z_1, \ldots, z_n$ are variables in \textit{dpool} which are arguments for $p_1, \ldots, p_n$, corresponding to $x_1, \ldots, x_n$

By (4), we can define a substitution

(5) $\tilde{\sigma} = \bigsqcup_{i=1}^{n} [z_i/x_i]$ 

By assumption,

$x_1, \ldots, x_n$ are mutually distinct variables

$i \neq j$ implies $\text{dom}([z_i/x_i]) \cap \text{dom}([z_j/x_j]) = \emptyset$

Thus $\tilde{\sigma}$ is well-defined.

\textsc{Ckpropdc} is called with arguments:

(6) $\tilde{c}_a = \Lambda_{i=1}^{n} c_{p_i}(z_i)$

and $c_{p_i}(z_i)$ is the constraint for the derivation of $p_i$ in \textit{dpool}

and $c_p(x_1, \ldots, x_n)\sigma = c_p(z_1, \ldots, z_n)$,

and $c_p(x_1, \ldots, x_n)$ is the constraint for $p_1, \ldots, p_n$ in $\varphi$

$d_1: (d_1(z_1); p_1(z_i))::\ldots::(d_n(z_n); p_n(z_n))::\text{nil}$

$d_1(z_1); p_1(z_i)$ is the derivation of $p_i$ in \textit{dpool}

$Q: q_1(y_1), \ldots, q_m(y_m)$ are as they appear in $\varphi$

$c_1: c_q(x_1, \ldots, x_n, y_1, \ldots, y_m)\sigma = c_q(z_1, \ldots, z_n, y_1, \ldots, y_m)$

and $c_q(x_1, \ldots, x_n, y_1, \ldots, y_m)$ is the constraint for $q_1, \ldots, q_m$ in $\varphi$

$\beta: b_1(u_1), \ldots, b_k(u_k)$ are base tuples as written in $\varphi_{net}$

$c_k: c_b(u_1, \ldots, u_k)$ is the constraint for base tuples $b_1, \ldots, b_k$ as written in $\varphi_{net}$

By Relation of variables in Derivation Pool,

(7) $\forall i \in \{1, 2, \ldots, n\}, z_i \subseteq z_1, z_i \subseteq z_n$

Next, we derive some facts about $\Sigma_b$ that will be used in all subcases.

By the semantics of \textsc{MergeLL} on Line 8, each $\sigma_{bd} \in \Sigma_b$ has form

(8) $\sigma_{bd} = \bigsqcup_{i=1}^{k} [z_{bd_i}/u_i]$

where $1 \leq i \leq \Sigma_{bd}$

and $z_{bd_i}$ are the arguments of $b_i$ as they appear in $d_\ell(z_{bd_i}); p_i(z_i)$ where $\ell \in [1, n]$

By the semantics of \textsc{Unity} and \textsc{MergeLL} on Line 8,

(9) $\Sigma_b = \sigma_1; \ldots; \sigma_{bd}; \text{nil}$ consists of

a list of all possible substitutions of arguments of $b_1, \ldots, b_k$ as written in $\varphi_{net}$,

for the arguments of $b_1, \ldots, b_k$ as they appear in

$(d_1(z_{bd_1}); p_1(z_i))::\ldots::(d_n(z_{bd_n}); p_n(z_n))::\text{nil}$

The true branch of the IF-ELSE statement from Lines 15-19 is taken. Using (7),

(A.1) $\exists \sigma_c \text{ s.t. } \Sigma_{bd} \vdash (\Lambda_{i=1}^{n} c_{p_i}(z_i)) \land c_p(z_1, \ldots, z_n) \land \Lambda_{i=1}^{n} c_b(u_1^i, \ldots, u_k^i)\sigma_{bd}\sigma_c$

we write $\sigma_c = \bigsqcup_{i=1}^{n} [t_i^c/z_{1i}^c],\text{ where } z_{1i}^c$ are the free variables in the above constraint.

By (A.1),

(A.2) $\forall i \in \{1, 2, \ldots, n\}, c_{p_i}(z_i)\sigma_c$

By (5),

(A.3) $\forall i \in \{1, 2, \ldots, n\}, (c_{p_i}(z_i), d_i(z_i); p_i(z_i)) \in \textit{dpool}(p_i)$

By Correctness of derivation pool construction (Lemma 1(2)) applied to (A.2) and (A.3),

(A.4) $\forall i \in \{1, 2, \ldots, n\}, \forall \sigma_{bd} \text{ s.t. } \sigma_{bd} \geq \sigma_c|z_i$, (we write $\sigma_{bd} = [t/d; z_i]$)

exists $B_i$ s.t. $B_i = \{b \mid b \text{ is a base tuple, } b \text{ occurs in } d_i(z_i); \sigma_{di}\}$,

and $\text{prog. } B_i \models d_i(z_i); \sigma_{di}; p_i(z_i)\sigma_{di}$ and $\models c(z_i)|\sigma_{di}$

By (A.4),

(A.5) $\forall i \in \{1, 2, \ldots, n\}$, we pick $\sigma_{di} \text{ s.t. } \sigma_{di} = \sigma_c|z_{di} \geq \sigma_c|z_{i1}$ and $B = \bigsqcup_{i=1}^{n} B_i$

By (4) and (A.4), we can define

(A.6) $\forall i \in \{1, 2, \ldots, n\}, \sigma_{pi} = ([z_i/x_i]|\sigma_{di}) = [t_i/x_i]$ 

By (A.4), (A.5), and (A.6),

$\forall i \in \{1, 2, \ldots, n\}$,

$d_i(z_{di})\sigma_{di}; p_i(z_i)|\sigma_{di} = d_i(z_{di})\sigma_{di}; p_i(x_i)|\sigma_{pi}$

By the above,

(A.7) $\forall i \in \{1, 2, \ldots, n\}$, $\text{prog. } B \models d_i(z_{di})\sigma_{di}; p_i(x_i)|\sigma_{pi}$
By (A.1),
\[ \models C_0(z_1, \ldots, z_n) \sigma_c \]
By the above,
\[ (A.8) \models C_p(x_1', \ldots, x_n') \sigma_{p_1} \ldots \sigma_{p_n} \]

Since the true branch of the If-ELSE statement from Lines 21-35 is taken, thus
\[ \exists j \in \{1, 2, \ldots, m\} \text{ s.t.} \]
\[ q_j \text{ does not appear in } (d_1(z_1p_1(z_1)): \ldots : (d_n(z_n)p_n(z_n)): \ldots : (d_n(z_n)p_n(z_n)): \ldots : (d_n(z_n)p_n(z_n)): \ldots : \text{nil} \]
By the above,
\[ (A.9) \forall \sigma_{q_1}, \ldots, \sigma_{q_m} \text{ for the variables } y_1, \ldots, y_m, \]
\[ \exists j \in \{1, 2, \ldots, m\} \text{ s.t.} \]
\[ (d_1(z_1p_1(z_1)): \ldots : (d_n(z_n)p_n(z_n)): \ldots : (d_n(z_n)p_n(z_n)): \ldots : (d_n(z_n)p_n(z_n)): \ldots : \text{nil} \]

Next, we show that \( B \models \psi_{net} \)
\[ (A.10) \text{Pick any } \sigma_{b_1}, \ldots, \sigma_{b_k} \text{ s.t. } \forall i \in \{1, 2, \ldots, k\}, \: b_i(u_i) \sigma_{b_i} \in B \]
By (A.4) and (A.10)
\[ (A.11) \: d_i(z_1p_1(z_1)) \sigma_{d_1} : \ldots : d_n(z_n)p_n(z_n) : \text{nil} \models \bigwedge_{i=1}^k c_i(u_i) \sigma_{b_i} \]
By (8) and (9) and (A.11)
\[ (A.12) \exists \sigma_{d_k} \in \Sigma_k \text{ s.t. } \forall i \in \{1, 2, \ldots, k\}, \: b_i(u_i) \sigma_{b_i} = b_i(u_i) \sigma_{d_k} \sigma_c \]
By (A.1),
\[ \models c_{d_k}(u_1, \ldots, u_k) \sigma_{d_k} \sigma_c \]
By (A.12),
\[ c_{d_k}(u_1, \ldots, u_k) \sigma_{d_k} \sigma_c = c_{b_1}(u_1, \ldots, u_k) \sigma_{b_1} \ldots \sigma_{b_k} \]
By the above,
\[ (A.13) \models c_{d_k}(u_1, \ldots, u_k) \sigma_{b_1} \ldots \sigma_{b_k} \]

By (A.5), (A.12) and (A.13),
\[ \text{prog. } B \not\models \psi \text{ and } B \models \psi_{net} \]

---

**Part 2, Case B:** CKPROP returns “invalid” on Line 34

The false branch of the If-ELSE statement from Lines 13-39 is taken, thus
\[ (B.1) \forall j \text{ where } j \in \{1, 2, \ldots, m\}, \]
\[ \exists \forall i \in \{1, 2, \ldots, n\} \text{ s.t. } q_j \text{ is in } d_i(z_1p_1(z_1)) \]

By the semantics of MERGELL on Line 26, each \( \sigma_{q_i} \in \Sigma_q \) has form
\[ (B.2) \exists \sigma_{d_i} \text{ s.t. } \sigma_{d_i} = \bigcup_{i=1}^n \{ t_{d_i}/z_1 \sigma_{d_i} \} \]
\[ \text{s.t. } a \bigwedge_{i=1}^L c_i(z_1, \ldots, z_n, y_1, \ldots, y_m) \sigma_{q_i} \bigwedge \bigwedge_{i=1}^L c_{d_i}(u_1, \ldots, u_k) \sigma_{d_i} \sigma_s \]

By (B.2),
\[ (B.3) \forall i \in \{1, 2, \ldots, n\}, \: c_{d_i}(z_1, \ldots, z_n) \sigma_s \]
By (6),
\[ (B.4) \forall i \in \{1, 2, \ldots, n\}, \: c_{d_i}(z_1, \ldots, z_n) \sigma_s \in \text{dpool}(p_i) \]

By Correctness of Derivation Pool construction (Lemma 1(2))
\[ (B.5) \forall i \in \{1, 2, \ldots, n\}, \]
\[ \forall \sigma_{d_i} \text{ s.t. } \sigma_{d_i} \geq \sigma_{d_i} \big|_{z_i} \text{ (let } \sigma_{d_i} = [t_{d_i}/z_1] \big) \]
\[ \exists B_i \text{ s.t. } B_i = \{ b \: b \text{ is a base tuple, } b \text{ appears in } d_i(z_1) \sigma_{d_i} \} \]
and \( \text{prog. } B_i \models d_i(z_1p_1(z_1)) \sigma_{d_i} \sigma_{d_i} = c_{d_i}(z_1, \ldots, z_n) \sigma_{d_i} \sigma_s \)

We pick \( \sigma_{d_i} \) such that \( \sigma_{d_i} = \sigma_{d_i} \big|_{z_i} \geq \sigma_{d_i} \big|_{z_i} \)
We define
\[ (B.6) \sigma_a = \sigma_{d_1} \ldots \sigma_{d_n} \]
We define
\[ (B.7) B = \bigcup_{i=1}^n B_i \]

By (4) and (B.5),
\[ (B.8) \forall i \in \{1, 2, \ldots, n\}, \: \sigma_{p_i} = ([z_1/x_1] \sigma_{d_i}) = [t_{d_i}/x_1] \]

By (B.8),
\[ \forall i \in \{1, 2, \ldots, n\}, \: d_i(z_1) \sigma_{d_i} \big|_{p_i(z_1)} \sigma_{d_i} = d_i(z_1) \sigma_{d_i} \big|_{p_i(x_1)} \sigma_{p_i} \]

By the above,
\[ (B.9) \forall i \in \{1, 2, \ldots, n\}, \: \text{prog. } B \models d_i(z_1p_1(z_1)) \sigma_{d_i} \sigma_{p_i} \]
By (B.3),
\[ c_p(z_1, \ldots, z_n) \sigma_a \]
Thus, using the above,
(B.10) \[ c_p(x_1, \ldots, x_n) \sigma_{p1} \ldots \sigma_{pn} \]
Pick any \( \sigma_{q1}, \ldots, \sigma_{qm} \) for \( y_1, \ldots, y_m \) s.t.
\[ \forall j \in \{1, 2, \ldots, m\}, \]
\[ d_1(z_{d1}) \sigma_{d1} p_1(x_1) \sigma_{p1} : \ldots : d_n(z_{dn}) \sigma_{dn} p_n(x_n) \sigma_{pn} : nil \models q_j(y_j) \sigma_q \]
By the property of \( \Sigma_q \)
(B.11) \[ \exists \sigma_{q\wedge} \in \Sigma_q \text{ where } \sigma_{q\wedge} = \bigwedge_{m \in \mathbb{N}} \left[ z_{qj}/y_j \right] \] s.t.
\[ \forall j \in \{1, 2, \ldots, m\}, q(y_j) \sigma_q \wedge = q(y_j) \sigma_{qj} \sigma_a \]
By (B.3),
\[ \models \neg c_q(z_1, \ldots, z_n, y_1, \ldots, y_m) \sigma_q \sigma_a \]
By (B.6) and (B.13),
\[ c_q(z_1, \ldots, z_n, y_1, \ldots, y_m) \sigma_q \sigma_a = c_q(x_1, \ldots, x_n, y_1, \ldots, y_m) \sigma_{p1} \ldots \sigma_{pn} \sigma_{q1} \ldots \sigma_{qm} \]
By the above,
(B.12) \[ \models \neg c_q(x_1, \ldots, x_n, y_1, \ldots, y_m) \sigma_{p1} \ldots \sigma_{pn} \sigma_{q1} \ldots \sigma_{qm} \]
By the above
\[ \text{prog, } B \not\models \phi \]
Next, we can prove \( B \models \phi_{net} \) in the same way as Case A.

## B. Definitions, Lemmas, and Theorems for Recursive Case

### B.1 Properties of the \( \Rightarrow_i \) relation

**Lemma 4.**
\[ \forall L \in \mathbb{N}, \]
\[ dpool \vdash D_{L-1}, \sigma_{L-1} \Rightarrow_i D_L, \sigma_L \]
implies
\[ D_L \text{ does not contain any (rec, \_)} \text{ leaves from the root up to its } L-1^{th} \text{ level.} \]

**Proof.**

**Proof by induction on \( L \).**

**Base Case: \( L = 0 \)**

The problem reduces to
\( (\star) \) \[ dpool \vdash D_0, \sigma_0 \Rightarrow_0 D_0, \sigma_0 \]
and showing that \( D_0 \) may not contain any (rec, \_) leaves on levels above the root.

\( D_0 \) has only the root (0\(^{th}\) level)
\( (\star) \) is vacuously true

**Inductive Case: \( L = K + 1 \)**

We proceed by induction on the derivation of \( d_K, \sigma_K \Rightarrow_i D_L, \sigma_L \)

**Case Base**

\[ \text{BASE } dpool \vdash d, \sigma \Rightarrow_0 d, \sigma \]

Since \( L \geq 1 \),
The lemma is vacuously true

**Case W\text{Kind}**

\[ dpool \vdash d, \sigma \Rightarrow_k d, \sigma \quad k < n \quad d \text{ does not contain (rec, \_) as subderivations} \]

By rule inversion,
(W1) $d = D_L$

$D_L$ does not contain any $(\text{rec, } _\cdot)$

By (W1),

$D_L$ does not contain any $(\text{rec, } _\cdot)$ leafs from the root up to its $K^\text{th}$ level

Case $\text{RNREC}$

$$\text{RNREC} \quad \forall i \in [1, n], \text{dpool} \vdash d_i, \sigma \leadsto_{k} d_i', \sigma_i \quad \sigma' = \bigcup_{i=1}^{n} \sigma_i$$

By rule inversion,

(R1) $D_L = (\text{rec, } p(\vec{x}), d_1: \ldots : d_n: \text{nil})$

$\forall i \in [1, n], \text{dpool} \vdash d_i, \sigma \leadsto_k d_i', \sigma_i$

Applying the induction hypothesis to (RN1),

(R2) $D_L$ does not contain any $(\text{rec, } _\cdot)$ leafs from the root up to its $K - 1^\text{th}$ level

By rule $\text{RNREC}$, (R1) and (R2),

$D_L$ does not contain any $(\text{rec, } _\cdot)$ leafs from the root up to its $K^\text{th}$ level

Case $\text{REC}$

$$\text{REC} \quad \text{dpool}(p) = (c_p, \Delta_p) \quad (c(\vec{z}'), d(\vec{z}); p(\vec{z})) \in \Delta_p \quad \vec{z}' = \text{fresh}(\vec{z}\setminus\vec{z}')$$

$$d(\vec{z}'; ((\vec{z}\setminus\vec{z}')[\vec{z}/\vec{z}])) \ni c(\vec{z})[\vec{z}'/(\vec{z}\setminus\vec{z})] = \bigcup_{i=1}^{n} \sigma_i$$

By rule inversion,

(R1) $D_L = d(\vec{z}'; ((\vec{z}\setminus\vec{z}')[\vec{z}/\vec{z}]))$

By function $\text{GEINDPOOL}$,

(R2) $d(\vec{z})$ is not of form $(\text{rec, } _\cdot)$

By (R1) and (R2),

(R3) $D_L$ does not contain any $(\text{rec, } _\cdot)$ at the root (its 0th level)

$\square$

B.2 Lemmas and Proof of Correctness Of Derivation Pool Construction (Recursive)

B.2.1 Lemmas used in the proof of Correctness Of Derivation Pool Construction

Lemma 5 (Sigma Extension Single Step). If $\text{dpool} \vdash d(\vec{x}_d); p(\vec{x}), \sigma_d \leadsto_k d'(\vec{x}_d'); p(\vec{x}), \sigma'_d$, then $\sigma' \geq \sigma$ and $\vec{x}_d' \supseteq \vec{x}_d$.

Proof:

[Proof by induction on the derivation of $d(\vec{x}_d); p(\vec{x}), \sigma \leadsto_k d(\vec{x}_d'), p(\vec{x}), \sigma'$]

Case BASE

| BASE | $\text{dpool} \vdash d, \sigma \leadsto_0 d, \sigma$ |

By rule inversion, we have:

(B1) $\sigma'_d = \sigma_d$

By (B1),

(B2) $\sigma'_d = \sigma_d \supseteq \sigma_d$

By rule inversion, we have:

(B3) $d'(\vec{x}_d') = d(\vec{x}_d)$

By (B3),

(B4) $\vec{x}_d' \supseteq \vec{x}_d$

By (B2) and (B4),

The conclusion holds

Case WKIND

| WKIND | $\text{dpool} \vdash d, \sigma \leadsto_k d, \sigma$ \quad $k < n$ \quad $d$ does not contain $(\text{rec, } _\cdot)$ as subderivations |

By rule inversion,

(W1) $\sigma'_d = \sigma_d$
By (W1),
(W2) \( \sigma'_d = \sigma_d \geq \sigma_d \)

By rule inversion,
(W3) \( d'(x'_d) = d(x_d) \)

By (W3),
(W4) \( x'_d = x_d \geq x_d \)

By (W2) and (W4),
The conclusion holds

\[
\text{Case RNREC}
\]

\[
\text{RNREC} \quad \forall i \in [1, n], \text{dpool} \vdash d_i, \sigma \rightsquigarrow_{k+1} d'_i, \sigma_i \quad \sigma' = \bigcup_{i=1}^n \sigma_i
\]

By rule inversion,
(N1) \( \sigma_d = \bigcup_{i=1}^n \sigma_i \)

By (N1) and the inductive Hypothesis,
(N2) \( \forall i \in [1, n], \sigma_i \geq \sigma_d \)

By (N1) and (N2),
(N3) \( \sigma'_d = \bigcup_{i=1}^n \sigma_i = \sigma_j \cup \bigcup_{i=1, i \neq j}^n \sigma_i \geq \sigma_d \)

By rule inversion,
(N4) \( d(x_d) = (rID, p(\bar{x}), d_1:::d_n::nil) \)
\( d'(\bar{x'}_d) = (rID, p(\bar{x}), d'_1:::d'_n::nil) \)

By the inductive hypothesis,
(N5) \( \forall i \in [1, n], \text{fe}(d_i) \supseteq \text{fe}(d_i) \)

By (N4) and (N5),
(N6) \( x'_d = \{ \bar{x}, \text{fe}(d_1), \ldots, \text{fe}(d_n) \} \supseteq \{ \bar{x}, \text{fe}(d_1), \ldots, \text{fe}(d_n) \} = x_d \)

By (N3) and (N6),
The conclusion holds

\[
\text{Case RREC}
\]

\[
dpool(p) = (c_p, \Delta_p) \quad (c(\bar{z'}), \Delta(\bar{z'}); p(\bar{x})) \in \Delta_p \quad \bar{z'}_d = \text{fresh}(\bar{z'}_d) \quad \bar{z'}_d \vdash \bar{z'}_d \subseteq \bar{z'}_d \quad \bar{z'}_d \subseteq (\bar{z'}_d)\]$

By rule inversion,
(R1) \( \sigma_d = \sigma \quad \sigma'_d = \sigma \cup \sigma' \)

By (R1),
(R2) \( \sigma'_d = \sigma \cup \sigma'' \geq \sigma = \sigma_d \)

By rule inversion,
(R3) \( d(x'_d):p(\bar{x}) = (\text{rec}, p(\bar{x})):p(\bar{x}), \quad d'(\bar{x'}_d):p(\bar{x}) = D(\bar{z}_d)[\bar{z'}_d/(\bar{z'}_d)] \)

By (R3),
(R4) \( x'_d = \bar{z}_d[\bar{z'}_d/(\bar{z'}_d)] \subseteq \bar{x} = x_d \)

By (R2) and (R4),
The conclusion holds

\[
\text{Lemma 6 (Sigma Extension).}
\forall L \in \text{N},
\text{dpool} \vdash d_L, \sigma_L
\vdash_{k+1} d_{k+1}, \sigma_{k+1}
\vdots
\vdash_{k+L} d_{k+L}, \sigma_{k+L}
\quad \text{implies}
\]
\[ \sigma_{\ell+L+1} \geq \sigma_{\ell} \text{ and } \text{fv}(d_{\ell+L+1}) \supseteq \text{fv}(d_{\ell}) \]

**Proof.**

**Base Case:** \( L = 0 \)

The theorem to be proved reduces to

(\(*\) \( \text{dpool} \vdash d_{\ell}, \sigma_{\ell} \leadsto_{\ell+1} d_{\ell+1}, \sigma_{\ell+1} \))

By Sigma Extension Single Step (Lemma 5),

(\(*\) holds)

**Base Case:** \( L = K + 1 \)

Assume the following:

(B1) \( \text{dpool} \vdash d_{\ell}, \sigma_{\ell} \leadsto_{\ell+1} d_{\ell+1}, \sigma_{\ell+1} \)

\[ \vdots \]

(B2) \( \sigma_{\ell+K+1} \geq \sigma_{\ell} \text{ and } \text{fv}(d_{\ell+K+1}) \supseteq \text{fv}(d_{\ell}) \)

By Sigma Extension Single Step (Lemma 5),

(\(B3\) \( \sigma_{\ell+L+1} \geq \sigma_{\ell+K+1} \text{ and } \text{fv}(d_{\ell+K+1}) \supseteq \text{fv}(d_{\ell+K+1}) \))

By (B2) and (B3),

(\(B4\) \( \sigma_{\ell+L+1} \geq \sigma_{\ell} \text{ and } \text{fv}(d_{\ell+L+1}) \supseteq \text{fv}(d_{\ell}) \))

By (B4),

The conclusion holds

\[ \square \]

**Lemma 7** (Unrolling Subderivations from Arbitrary Depth).

\( \forall K \in \mathbb{N}, \forall L \geq 1, \forall i \in \{1, 2, \ldots, n\}, \text{dpool} \vdash d_{(i)K}, \sigma_K \leadsto_{K} d_{(i)K+1}, \sigma_{(i)K+1} \)

\[ \vdots \]

\( \leadsto_{K+\ell} d_{(i)K+\ell+1}, \sigma_{(i)K+\ell+1} \)

\[ \vdots \]

\( \leadsto_{K+L} d_{(i)K+L+1}, \sigma_{(i)K+L+1} \)

and \( i \neq j \) implies \( \text{fv}(d_{(i)K+L+1}) \cap \text{fv}(d_{(j)K+L+1}) = \emptyset \)

and \( \bigcup_{i=1}^{n} \sigma_{(i)K+L+1} \) is well-defined

implies

\( \text{dpool} \vdash (rID, p(\vec{x}), d_{(1)K} \ddots : d_{(n)K} \ast nil), \sigma_K \leadsto_{K+1} (rID, p(\vec{x}), d_{(1)K+1} \ddots : d_{(n)K+1} \ast nil), \bigcup_{i=1}^{n} \sigma_{(i)K+1} \)

\[ \vdots \]

\( \leadsto_{K+\ell+1} (rID, p(\vec{x}), d_{(1)K+\ell+1} \ddots : d_{(n)K+\ell+1} \ast nil), \bigcup_{i=1}^{n} \sigma_{(i)K+\ell+1} \)

\[ \vdots \]

\( \leadsto_{K+L+1} (rID, p(\vec{x}), d_{(1)K+L+1} \ddots : d_{(n)K+L+1} \ast nil), \bigcup_{i=1}^{n} \sigma_{(i)K+L+1} \)

**Proof.**

Pick any \( K \in \mathbb{N}. \)

**Prove the lemma by induction on \( L, \) the number of steps the derivation at depth \( K \) is unrolled to.**

**Base Case:** \( L = 0 \)

The lemma to be proved reduces to:

(\(*\) \( \forall i \in \{1, 2, \ldots, n\}, \text{dpool} \vdash d_{(i)K}, \sigma_K \leadsto_{K} d_{(i)K+1}, \sigma_{(i)K+1} \)
Combining (8) and (14), we derive our goal:

By (4) and (10),

By Sigma Extension (Lemma 6),

Assume the following:

1. \( \forall i \in \{1, 2, \ldots, n\}, \)
   \[ dpool \vdash d_{(i)K}, \sigma_K \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
   \[ \cdots \]
\( d_{pool} \vdash (rID, p(\vec{x}), d_{(1)K}::\ldots::d_{(n)K}::nil), \sigma_K \)
\(~\Rightarrow_{K+1} (rID, p(\vec{x}), d_{(1)K+1}::\ldots::d_{(n)K+1}::nil), \bigcup_{i=1}^{n} \sigma_{(i)K+1} \)
\(~\Rightarrow_{K+\ell+1} (rID, p(\vec{x}), d_{(1)K+\ell+1}::\ldots::d_{(n)K+\ell+1}::nil), \bigcup_{i=1}^{n} \sigma_{(i)K+\ell+1} \)
\(~\Rightarrow_{K+\lambda+1} (rID, p(\vec{x}), d_{(1)K+\lambda+1}::\ldots::d_{(n)K+\lambda+1}::nil), \bigcup_{i=1}^{n} \sigma_{(i)K+\lambda+1} \)
\(~\Rightarrow_{K+\mu+1} (rID, p(\vec{x}), d_{(1)K+\mu+1}::\ldots::d_{(n)K+\mu+1}::nil), \bigcup_{i=1}^{n} \sigma_{(i)K+\mu+1} \)
Lemma 8 (Unrolling Subderivations from Recursive Predicate).

\[ \text{dpool} \vdash (\text{rec}, p(\overline{x})), \sigma \]
\[ \sim_0 (\text{rec}, p(\overline{x})), \sigma \]
\[ \sim_1 (rID, p(x), d_{(1)}0:::d_{(n)}0::nil), \sigma \cup \bigcup_{i=1}^{n} \sigma_{(i)0} \]
and \( \forall i \in \{1, 2, \ldots, n\} \),
\[ \text{dpool} \vdash d_{(i)}0, \sigma_{(i)0} \]
\[ \sim_0 d_{(i)}0, \sigma_{(i)0} \]
\[ \sim_1 d_{(i)}1, \sigma_{(i)1} \]
\[ \ldots \]
\[ \sim_L d_{(i)L+1}, \sigma_{(i)L+1} \]
\[ \text{implies} \]
\[ \text{dpool} \vdash (\text{rec}, p(\overline{x})), \sigma \]
\[ \sim_0 (\text{rec}, p(\overline{x})), \sigma \]
\[ \sim_1 (rID, p(x), d_{(1)}0:::d_{(n)}0::nil), \sigma \cup \bigcup_{i=1}^{n} \sigma_{(i)0} \]
\[ \ldots \]
\[ \sim_0 (\text{rec}, p(\overline{x})), \sigma \]
\[ \sim_1 (rID, p(x), d_{(1)}0:::d_{(n)}0::nil), \sigma \cup \bigcup_{i=1}^{n} \sigma_{(i)0} \]
\[ \ldots \]
\[ \sim_L (rID, p(x), d_{(1)L+1}:::d_{(n)L+1}::nil), \sigma \cup \bigcup_{i=1}^{n} \sigma_{(i)L+1} \]

Proof.

Assume the following:

1. \( \text{dpool} \vdash (\text{rec}, p(\overline{x})), \sigma \)
2. \( \sim_0 (\text{rec}, p(\overline{x})), \sigma \)
3. \( \sim_1 (rID, p(x), d_{(1)}0:::d_{(n)}0::nil), \sigma \cup \bigcup_{i=1}^{n} \sigma_{(i)0} \)
4. \( \ldots \)
5. \( \sim_L (rID, p(x), d_{(1)L+1}:::d_{(n)L+1}::nil), \sigma \cup \bigcup_{i=1}^{n} \sigma_{(i)L+1} \)

By the algorithm,
\( f_{\text{v}}(d_{(1)L+1}), \ldots, f_{\text{v}}(d_{(n)L+1}) \) are fresh.

By the above,
\( (\sigma \cup \bigcup_{i=1}^{n} \sigma_{(i)0}) \cup (\sigma_{(i)L+1}) \) is well-defined.

By the above, we apply Weakening (Lemma 11) to obtain:

6. \( \forall i \in \{1, 2, \ldots, n\} \),
7. \( \text{dpool} \vdash d_{(i)}0, (\sigma \cup \bigcup_{i=1}^{n} \sigma_{(i)0}) \cup \sigma_0 \)
8. \( \ldots \)
9. \( \sim_1 d_{(i)}1, (\sigma \cup \bigcup_{i=1}^{n} \sigma_{(i)0}) \cup \sigma_{(i)1} \)
10. \( \ldots \)
11. \( \sim_L d_{(i)L+1}, (\sigma \cup \bigcup_{i=1}^{n} \sigma_{(i)0}) \cup \sigma_{(i)L+1} \)

By (3), we can use Unrolling Subderivations from Arbitrary Depth (Lemma 7) to obtain:

12. \( \text{dpool} \vdash (rID, p(x), d_{(1)}0:::d_{(n)}0::nil), \sigma \cup \bigcup_{i=1}^{n} \sigma_{(i)0} \)
13. \( \sim_1 (rID, p(x), d_{(1)}0:::d_{(n)}0::nil), \sigma \cup \bigcup_{i=1}^{n} \sigma_{(i)0} \)
14. \( \ldots \)
15. \( \sim_L (rID, p(x), d_{(1)L+1}:::d_{(n)L+1}::nil), \sigma \cup \bigcup_{i=1}^{n} \sigma_{(i)L+1} \)

Combining (1) and (4), we have:

16. \( \text{dpool} \vdash (\text{rec}, p(\overline{x})), \sigma \)
17. \( \sim_0 (\text{rec}, p(\overline{x})), \sigma \)
18. \( \sim_1 (rID, p(x), d_{(1)}0:::d_{(n)}0::nil), \sigma \cup \bigcup_{i=1}^{n} \sigma_{(i)0} \)
19. \( \ldots \)
20. \( \sim_L (rID, p(x), d_{(1)L+1}:::d_{(n)L+1}::nil), \sigma \cup \bigcup_{i=1}^{n} \sigma_{(i)L+1} \)
Lemma 9 (Unrolling Subderivations).  
\[ \forall i \in \{1, 2, \ldots, n\}, \quad dpool \vdash d_{i, 0}, \sigma_0 \]

\[ \ldots \Rightarrow d_{i, \ell+1}, \sigma_{\ell+1} \]

\[ \ldots \Rightarrow d_{i, L+1}, \sigma_{L+1} \]

and \( i \neq j \) implies \( \text{fv}(d_{i, L+1}) \cap \text{fv}(d_{j, L+1}) = \emptyset \)

implies

\[ dpool \vdash (rID, p(\vec{x}), d_{1, 0}, \ldots, d_{n, 0}, nil), \sigma_0 \]

\[ \ldots \Rightarrow (rID, p(\vec{x}), d_{1, \ell+1}, \ldots, d_{n, \ell+1}, nil), \bigcup_{i=1}^n \sigma_{\ell+1} \]

\[ \ldots \Rightarrow (rID, p(\vec{x}), d_{1, L+1}, \ldots, d_{n, L+1}, nil), \bigcup_{i=1}^n \sigma_{L+1} \]

Proof. By Unrolling Subderivations from Arbitrary Depth (Lemma 7), setting \( K = 0 \), the lemma holds.

Lemma 10 (Weakening single step). If \( dpool \vdash d_{\ell}, \sigma_{\ell} \Rightarrow d_{\ell+1}, \sigma_{\ell+1} \) and \( \bar{\sigma} \cup \sigma_{\ell+1} \) is well-defined, then \( dpool \vdash d_{\ell}, \bar{\sigma} \cup \sigma_{\ell} \Rightarrow d_{\ell+1}, \bar{\sigma} \cup \sigma_{\ell+1} \).

Proof. 
By induction on the derivation of \( dpool \vdash d_{\ell}, \sigma_{\ell} \Rightarrow d_{\ell+1}, \sigma_{\ell+1} \)

<table>
<thead>
<tr>
<th>Case</th>
<th>Base</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ dpool \vdash d, \sigma \Rightarrow_0 d, \sigma ]</td>
<td></td>
</tr>
</tbody>
</table>

By rule inversion,

(B1) \( \ell = 0 \)

\[ d_0 = d = d_{\ell+1} \]

\[ \sigma_0 = \sigma = \sigma_{\ell+1} \]

Applying Base.

(B2) \( \forall \sigma', dpool \vdash d_\ell, \sigma' \Rightarrow_0 d_{\ell+1}, \sigma' \)

By assumption,

(B3) \( \bar{\sigma} \cup \sigma_{\ell+1} \) is well-defined

By (B2) and (B3),

(B5) \( dpool \vdash d_\ell, \bar{\sigma} \cup \sigma_{\ell+1} \Rightarrow_0 d_{\ell+1}, \bar{\sigma} \cup \sigma_{\ell+1} \)

By (B1) and (B5),

(B6) \( dpool \vdash d_\ell, \bar{\sigma} \cup \sigma_{\ell} \Rightarrow_0 d_{\ell+1}, \bar{\sigma} \cup \sigma_{\ell+1} \)

the conclusion holds

<table>
<thead>
<tr>
<th>Case</th>
<th>WkInd</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ dpool \vdash d, \sigma \Rightarrow_k d, \sigma ]</td>
<td></td>
</tr>
</tbody>
</table>

By rule inversion,

(W1) \( \ell = n \)

\[ d_\ell = d = d_{\ell+1} \]

\[ \sigma_{\ell+1} = \sigma = \sigma_\ell \]

\( d \) does not contain \((\text{rec}, \_\_\) as subderivations

\( k < n \)

By assumption,

(W2) \( \bar{\sigma} \cup \sigma_{\ell+1} \) is well-defined

By (W1) and (W2), we can apply the Induction Hypothesis to obtain:

(W3) \( dpool \vdash d, \bar{\sigma} \cup \sigma \Rightarrow_k d, \bar{\sigma} \cup \sigma \)

Using (W1) and (W3), we can apply WkInd to obtain:

(W4) \( dpool \vdash d, \bar{\sigma} \cup \sigma \Rightarrow_\ell d, \bar{\sigma} \cup \sigma \)

By (W1), the above can be re-written as

(W5) \( dpool \vdash d_\ell, \bar{\sigma} \cup \sigma_{\ell} \Rightarrow_\ell d_\ell, \sigma_{\ell+1} \cup \bar{\sigma} \)
**Case R\(_{\text{REC}}\)**

\[
\frac{\forall i \in [1, n], \text{dpool} \vdash d_i, \sigma \rightsquigarrow d'_i, \sigma'}{\text{dpool} \vdash (\text{rID}, p(\bar{x}), d_1::\ldots::d_n::\text{nil}), \sigma \rightsquigarrow_{k+1} (\text{rID}, p(\bar{x}), d'_1::\ldots::d'_n::\text{nil}), \sigma'}
\]

By rule inversion,
(RN1) \( \ell = k + 1 \)
\[
\begin{align*}
\text{de} &= (\text{rID}, p(\bar{x}), d_1::\ldots::d_n::\text{nil}) \\
\text{de}_{\ell+1} &= (\text{rID}, p(\bar{x}), d'_1::\ldots::d'_n::\text{nil}) \\
\sigma_{\ell} &= \sigma \\
\sigma_{\ell+1} &= \sigma' = \bigcup_{i=1}^{n} \sigma_i
\end{align*}
\]

By assumption,
\( \sigma \cup \bigcup_{i=1}^{n} \sigma_i \) is well-defined

By the above,
(RN2) \( \forall i \in \{1, 2, \ldots, n\}, \sigma \cup \sigma_i \) is well-defined

Using (RN2), we can apply the Induction Hypothesis to obtain:
(RN3) \( \forall i \in \{1, 2, \ldots, n\}, \text{dpool} \vdash \text{de}_i, \sigma \rightsquigarrow d'_i, \sigma \cup \sigma_i \)

Using (RN3), we apply \( \text{R}\(_{\text{REC}}\) \) and derive our goal:
(RN4) \( \text{dpool} \vdash (\text{rID}, p(\bar{x}), d_1::\ldots::d_n::\text{nil}), \sigma \cup \sigma \rightsquigarrow_{k+1} (\text{rID}, p(\bar{x}), d'_1::\ldots::d'_n::\text{nil}), \sigma \cup \sigma' \)

\[\begin{array}{c}
\text{dpool}(p) = (c_p, \Delta_p) \\
\text{R}_{\text{REC}} \quad (c(\bar{z}_d), d(\bar{z}_d):p(\bar{z})) \in \Delta_p \\
\quad \bar{z}_d' = \text{fresh}(\bar{z}_d \setminus \bar{z}) \\
\text{dpool} \vdash (\text{rec}, p(\bar{x})), \sigma \rightsquigarrow_{1} d(\bar{z}_d')/\bar{z}'[\bar{x}/\bar{z}], \sigma \cup \sigma'
\end{array}\]

By rule inversion,
(R1) \( \ell = 1 \)
\[
\begin{align*}
\text{de} &= (\text{rec}, p(\bar{x})) \\
\text{de}_{\ell+1} &= d(\bar{z}_d')/\bar{z}'[\bar{x}/\bar{z}] \\
\sigma_{\ell} &= \sigma \\
\sigma_{\ell+1} &= \sigma \cup \sigma' \\
\text{dpool}(p) &= (c_p, \Delta_p) \\
(c(\bar{z}_d), d(\bar{z}_d):p(\bar{z})) &\in \Delta_p \\
\bar{z}_d' &= \text{fresh}(\bar{z}_d \setminus \bar{z}) \\
\vdash c(\bar{z}_d')[\bar{z}'[\bar{x}/\bar{z}]](\sigma \cup \sigma')
\end{align*}
\]

By assumption,
\( \bar{\sigma} \cup (\sigma \cup \sigma') \) is well-defined

By the above,
(R2) \( \bar{\sigma} \cup \sigma \) is well-defined

By (R1),
(R3) \( \vdash c(\bar{z}_d')[\bar{z}'[\bar{x}/\bar{z}]](\bar{\sigma} \cup \sigma \cup \sigma') \)

Using (R1) and (R3), we apply \( \text{R}_\text{REC} \) to obtain our goal:
(R4) \( \text{dpool} \vdash (\text{rec}, p(\bar{x})), \bar{\sigma} \cup \sigma \rightsquigarrow_{1} d(\bar{z}_d')[\bar{z}'[\bar{x}/\bar{z}]], \bar{\sigma} \cup (\sigma \cup \sigma') \)

\[\Box\]

**Lemma 11** (Weakening).
\( \forall \bar{\sigma}, \)\( \text{dpool} \vdash d_0, \sigma_0 \)
\[
\begin{array}{c}
\vdots \\
\rightsquigarrow_{\ell} d_{\ell+1}, \sigma_{\ell+1} \\
\vdots \\
\rightsquigarrow_{L+1} d_{L+1}, \sigma_{L+1}
\end{array}
\]
and \( \bar{\sigma} \cup \sigma_{L+1} \) is well-defined implies
\( \text{dpool} \vdash d_0, \bar{\sigma} \cup \sigma_0 \)
\[
\begin{array}{c}
\vdots \\
\rightsquigarrow_{\ell} d_{\ell+1}, \bar{\sigma} \cup \sigma_{\ell+1} \\
\vdots
\end{array}
\]
Proof. Pick any \( \bar{\sigma} \).

Assume the following:
(1) \( \text{dpool} \vdash d_0, \sigma_0 \rightarrow_\ell d_{\ell+1}, \sigma_{\ell+1} \)
(2) \( \bar{\sigma} \cup \sigma_{\ell+1} \) is well-defined

By Sigma Extension ((Lemma 6)),
\( \forall \ell \in \{0, 1, \ldots, L\}, \sigma_\ell \leq \sigma_{\ell+1} \)

By (2) and the above,
(3) \( \forall \ell \in \{0, 1, \ldots, L\}, \bar{\sigma} \cup \sigma_\ell \) is well-defined

By (1) and (3), using Weakening Single Step (Lemma 10),
(4) \( \forall \ell \in \{0, 1, \ldots, L\}, \bar{\sigma} \cup \sigma_\ell \rightarrow_\ell d_{\ell+1}, \bar{\sigma} \cup \sigma_{\ell+1} \)

By (4), we obtain our desired goal:
(5) \( \text{dpool} \vdash d_0, \bar{\sigma} \rightarrow_0 d_0, \bar{\sigma} \rightarrow_\ell \rightarrow_\ell d_{\ell+1}, \bar{\sigma} \cup \sigma_{\ell+1} \)

Lemma 12 (Substitution single step). If \( \text{dpool} \vdash d_\ell(x_\ell), \sigma_\ell \rightarrow_\ell d_{\ell+1}(x_\ell, x_{\ell+1}), \sigma_{\ell+1} \) and \( \bar{z}_\ell \) are fresh variables that correspond to \( x_\ell \), then given any fresh variables \( \bar{z}_{\ell+1} \) for \( x_{\ell+1} \), \( \text{dpool} \vdash d_\ell(\bar{z}_\ell), [\bar{x}_\ell/\bar{z}_\ell] \sigma_\ell \rightarrow_\ell d_{\ell+1}(\bar{z}_\ell, z_{\ell+1}), [\bar{x}_\ell/\bar{z}_\ell][x_{\ell+1}/z_{\ell+1}] \sigma_{\ell+1} \).

Case BASE

By rule inversion,
(B1) \( \ell = 0 \)
\( d_\ell(x_0) = d = d_{\ell+1}(x_\ell, x_{\ell+1}) \)
\( \sigma_\ell = \sigma = \sigma_{\ell+1} \)

By (B1),
\( x_{\ell+1} = \emptyset \)

By the above,
(B2) \( z_{\ell+1} = \emptyset \)

By (B1),
(B3) \( [\bar{x}_\ell/\bar{z}_\ell] \sigma_\ell = [\bar{x}_\ell/\bar{z}_\ell] \sigma = [\bar{x}_\ell/\bar{z}_\ell][x_{\ell+1}/z_{\ell+1}] \sigma_{\ell+1} \)

Applying CASE BASE,
(B4) \( \text{dpool} \vdash d, [\bar{x}_\ell/\bar{z}_\ell] \sigma \rightarrow_0 d, [\bar{x}_\ell/\bar{z}_\ell] \sigma \)
The conclusion holds

Case WkIND

<table>
<thead>
<tr>
<th>WkIND</th>
<th>\text{dpool} \vdash d, \sigma \rightarrow_k d, \sigma</th>
<th>k &lt; n</th>
<th>d \text{ does not contain(\text{rec}, _)} \text{as subderivations}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>\text{dpool} \vdash d, \sigma \rightarrow_\ell d, \sigma</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

By rule inversion,
(W1) \( \ell = n \)
\( d_\ell(x_\ell) = d = d_{\ell+1}(x_\ell, x_{\ell+1}) \)
\( \sigma_\ell = \sigma = \sigma_{\ell+1} \)
By (B1),
\[ x_{i+1} = 0 \]
By the above,
\[ (B2) \ z_{i+1} = 0 \]

By (W1),
\[ (W3) \ [\bar{x}/z][\sigma] = [\bar{x}/z][z_{i+1}]/z_{i+1}]\sigma_{i+1} \]

Applying WKind,
\[ (W4) \ dpool \vdash d, [\bar{x}/z][\sigma] \rightsquigarrow_n d, [\bar{x}/z][\sigma] \]

The conclusion holds

---

### Case \text{Rnrec}

\[ \forall i \in [1, n], \ dpool \vdash d_i, \sigma \rightsquigarrow d_i', \sigma_i \quad \sigma' = \bigcup_{i=1}^n \sigma_i \]

By rule inversion,

(N1) \[ \ell = k + 1 \]
\[ d(\bar{x}) = (rID, p(\bar{x}), d_1:\ldots:d_n:\nil) \]
\[ d_{i+1}(\bar{x}, x_{i+1}) = (rID, p(\bar{x}), d_1':\ldots:d'_n:\nil) \]
\[ \sigma_\ell = \sigma \]
\[ \sigma_{i+1} = \sigma' = \bigcup_{i=1}^n \sigma_i \]
\[ x_{i+1} = \bigcup_{i=1}^n fv(d_i) \backslash fv(d_i') \]
\[ x_{\bar{x}} = \{ x, \bar{x}, \ldots, \bar{x}_n \} \]

By (N1), letting \( \bar{z}_i \) be fresh variables corresponding to \( \bar{x}_i \),

(N2) Define \( \bar{z}_i = \{ \bar{z}_1', \bar{z}_2', \ldots, \bar{z}_n' \} \),

where \( \bar{z}_1', \ldots, \bar{z}_n' \) correspond to the free variables of \( d_1, \ldots, d_n \)

By (N1) and (N2), we apply the Induction Hypothesis to obtain:

(N2) \[ \forall i \in [1, 2, \ldots, n], \]
Given any fresh variables \( \bar{z}_i' \) for \( fv(d_i) \backslash fv(d_i') \),
\[ dpool \vdash d_i(\bar{z}_i' /fv(d_i)), [fv(d_i') /\bar{z}_i][\sigma_\ell] \]
\[ \rightsquigarrow_n d_i'([\bar{z}_i' /fv(d_i')])\] \[ ([\bar{z}_i' /fv(d_i') \backslash fv(d_i)]\sigma_i) \]

Pick any \( \bar{z}_{i+1}' \) which are fresh variables corresponding to \( x_{i+1} \).

By (N1), then \( \bar{z}_{i+1}' \) are fresh variables for \( d_1', \ldots, d_n' \).

Using (N2), we apply Rnrec to obtain:

(N3) \[ dpool \vdash (rID, p(\bar{z}), d_1[\bar{z}_1', \ldots, d_n[\bar{z}_n'/fv(d_n')]):\ldots,d_n[\bar{z}_n'/fv(d_n')]:\nil), [fv(\bar{z}) /\bar{x}_{\bar{x}}][\sigma_\ell] \]
\[ \rightsquigarrow_{k+1} (rID, p(\bar{z}), d_1[\bar{z}_1', \ldots, d_n[\bar{z}_n'/fv(d_n')]):\ldots,d_n[\bar{z}_n'/fv(d_n')]:\nil), [fv(\bar{z}) /\bar{x}_{\bar{x}}][\bigcup_{i=1}^n [\bar{z}_i'/fv(d_i') \backslash fv(d_i)]\sigma_i] \]

By (N1), we can rewrite (N3) as:

(N4) \[ dpool \vdash (rID, p(\bar{z}), d_1[\bar{z}_1'/fv(d_1')]:\ldots,d_n[\bar{z}_n'/fv(d_n')]:\nil), [fv(\bar{z}) /\bar{x}_{\bar{x}}][\sigma_\ell] \]
\[ \rightsquigarrow_{k+1} (rID, p(\bar{z}), d_1[\bar{z}_1'/fv(d_1')]:\ldots,d_n[\bar{z}_n'/fv(d_n')]:\nil), [fv(\bar{z}) /\bar{x}_{\bar{x}}][\bigcup_{i=1}^n [\bar{z}_i'/fv(d_i') \backslash fv(d_i)]\sigma_i] \]

The conclusion holds

---

### Case \text{Rrec}

\[ dpool(p) = (c_p, \Delta_p) \quad (c(\bar{z}_i'), d(\bar{z}_i'); p(\bar{z})) \in \Delta_p \quad \bar{z}_i' = \text{fresh} (\bar{z}_i \backslash \bar{z}) \]

\[ \sigma' = c(\bar{z}_i' \backslash (\bar{z}_i \backslash \bar{z}))(\bar{x} / \bar{z})(\sigma \cup \sigma') \]

By rule inversion,

(R1) \[ \ell = 1 \]
\[ d(x) = (\text{rec}, p(\bar{x})) \]
\[ d_{i+1}(\bar{x}, x_{i+1}) = d(\bar{z}_i') \]
\[ \sigma_\ell = \sigma \]
\[ \sigma_{i+1} = \sigma \cup \sigma' \]
\[ dpool(p) = (c_p, \Delta_p) \]
\[ (c(\bar{z}_i'), d(\bar{z}_i'); p(\bar{z})) \in \Delta_p \]
\[ \bar{z}_i' = \text{fresh} (\bar{z}_i \backslash \bar{z}) \]
\[ \vdash c(\vec{z}_c)[\vec{z}_c'/\{\vec{x}\}](\bar{x}/\vec{z})(\sigma \cup \sigma') \]

By (R1),
(R2) \( \vec{x}_\ell = \bar{x} \)
\( \vec{z}_i \) are fresh variables for \( \bar{x} \)
\( \vec{x}_{\ell + 1} = \vec{z}_{d}' \)

(R3) Pick any \( \vec{z}_{\ell + 1} \) which are fresh variables for \( \vec{z}_d' \).
Then \( \vec{z}_{\ell + 1} = fresh(\vec{z}_d \setminus \vec{z}) \)

By (R1), (R2), and (R3), we have:
(R4) \[ \vdash c(\vec{z}_c)[\vec{z}_{\ell + 1}/\{\vec{x}\}][\vec{z}_i'/\vec{z}], \bar{\sigma} \]
where \( \bar{\sigma} = [\bar{x}/\vec{z}][(\vec{z}_d \setminus \vec{z})/\vec{z}_{\ell + 1}]\sigma \cup \sigma' \)

By (R1) and (R4), we apply \textsc{Rec} to obtain:
(R5) \[ dpool \vdash (\text{rec}, p(\vec{z}_i)), [\bar{x}/\vec{z}]\sigma \Downarrow_1 d(x_\ell)[\vec{z}_{\ell + 1}/\{\vec{x}\}][\vec{z}_i'/\vec{z}], \bar{\sigma} \]
Lemma 13 (Substitution),
\[ dpool \vdash d_0(\vec{x}_0), \sigma_0 \]
\[ \vdash_0 d_1(\vec{z}_0), \sigma_1 \]
\[ \ldots \]
\[ \vdash_{\ell} d_{\ell+1}(\vec{x}_0, \vec{x}_1, \ldots, \vec{x}_\ell), \sigma_{\ell+1} \]
\[ \ldots \]
\[ \vdash_L d_{L+1}(\vec{x}_0, \vec{x}_1, \ldots, \vec{x}_L), \sigma_{L+1} \]
and \( \vec{z}_0 \) are fresh variables corresponding to \( \vec{x}_0 \), implies
\[ \forall \vec{z}_1, \ldots, \vec{z}_L \text{ which correspond to fresh variables for } \vec{x}_1, \ldots, \vec{x}_L, \]
\[ dpool \vdash d_0(\vec{z}_0), [\sigma_0(\vec{z}_0)/\vec{x}_0] \]
\[ \vdash_0 d_1(\vec{z}_0), [\sigma_1(\vec{z}_0)/\vec{x}_0] \]
\[ \ldots \]
\[ \vdash_{\ell} d_{\ell+1}(\vec{z}_0, \vec{z}_1, \ldots, \vec{z}_\ell), [\sigma_{\ell+1}(\vec{x}_0, \vec{x}_1, \ldots, \vec{x}_\ell)/\vec{z}_0, \vec{z}_1, \ldots, \vec{z}_\ell] \]
\[ \ldots \]
\[ \vdash_L d_{L+1}(\vec{z}_0, \vec{z}_1, \ldots, \vec{z}_L), [\sigma_{L+1}(\vec{x}_0, \vec{x}_1, \ldots, \vec{x}_L)/\vec{z}_0, \vec{z}_1, \ldots, \vec{z}_L] \]

Proof:
By induction on \( L \).

Base Case: \( L = 0 \)

The lemma to be proved reduces to:
1. \( dpool \vdash d_0(\vec{x}_0), \sigma_0 \vdash_0 d_1(\vec{z}_0), \sigma_1 \)
and \( \vec{z}_0 \) are fresh variables corresponding to \( \vec{x}_0 \), implies
\[ dpool \vdash d_0(\vec{z}_0), [\sigma_0(\vec{z}_0)/\vec{x}_0] \vdash_0 d_1(\vec{z}_0), [\sigma_1(\vec{z}_0)/\vec{x}_0] \]

By rule inversion,
(B1) \( \sigma_1 = \sigma_0 \)
\[ d_0(\vec{x}_0) = d_1(\vec{z}_0) \]

By (B1),
(B2) \( d_0(\vec{z}_0) = d_1(\vec{z}_0) \)
\[ [\sigma_0(\vec{z}_0)/\vec{x}_0] = [\sigma_1(\vec{z}_0)/\vec{x}_0] \]

By (B2), we apply Base to obtain:
(B3) \( dpool \vdash d_0(\vec{z}_0), [\sigma_0(\vec{z}_0)/\vec{x}_0] \vdash_0 d_1(\vec{z}_0), [\sigma_1(\vec{z}_0)/\vec{x}_0] \]
The conclusion holds

Inductive Case: \( L = \lambda + 1 \)

Assume the following:
(I1) \( dpool \vdash d_0(\vec{x}_0), \sigma_0 \)
\[ \vdash_0 d_1(\vec{x}_0), \sigma_1 \]
\[ \ldots \]
\[ \vdash_{\ell} d_{\ell+1}(\vec{x}_0, \vec{x}_1, \ldots, \vec{x}_\ell), \sigma_{\ell+1} \]
\[ \ldots \]
\[ \vdash_{\lambda} d_{\lambda+1}(\vec{x}_0, \vec{x}_1, \ldots, \vec{x}_{\lambda}), \sigma_{\lambda+1} \]
\[ \vdash_L d_{L+1}(\vec{x}_0, \vec{x}_1, \ldots, \vec{x}_L), \sigma_{L+1} \]
(12) \( \vec{z}_0 \) are fresh variables corresponding to \( \vec{x}_0 \)

Pick any \( \vec{z}_1, \ldots, \vec{z}_L \) which correspond to fresh variables for \( \vec{x}_1, \ldots, \vec{x}_L \).

By (I1),
(I3) \( dpool \vdash d_0(\vec{z}_0), \sigma_0 \)
\[ \vdash_0 d_1(\vec{z}_0), \sigma_1 \]
\[ \ldots \]
\[ \vdash_{\ell} d_{\ell+1}(\vec{z}_0, \vec{x}_1, \ldots, \vec{x}_\ell), \sigma_{\ell+1} \]
\[ \ldots \]
\[ \vdash_{\lambda} d_{\lambda+1}(\vec{z}_0, \vec{x}_1, \ldots, \vec{x}_{\lambda}), \sigma_{\lambda+1} \]

By (I2) and (I3), we apply the Induction Hypothesis to obtain:
(14) \( dpool \vdash d_0(\vec{z}_0), [\sigma_0(\vec{z}_0)/\vec{x}_0] \)
\[ \vdash_0 d_1(\vec{z}_0), [\sigma_1(\vec{z}_0)/\vec{x}_0] \]
\[ \ldots \]
By inspection,

\[ d_{L+1}(z_0, z_1, \ldots, z_l), [\sigma_{L+1}(x_0, x_1, \ldots, x_l)] / z_0, z_1, \ldots, z_l \]

\[ \ldots \]

\[ d_{L+1}(z_0, z_1, \ldots, z_l), [\sigma_{L+1}(x_0, x_1, \ldots, x_l)] / z_0, z_1, \ldots, z_l \]

By (11),

(15) \( dpool \vdash d_{L+1}(z_0, z_1, \ldots, z_l), \sigma_{L+1} \sim_L d_{L+1}(x_0, x_1, \ldots, x_l), \sigma_{L+1} \)

By (15), we apply Substitution Single Step (Lemma 12) to obtain:

(16) \( dpool \vdash d_{L+1}(z_0, z_1, \ldots, z_l), [\sigma_{L+1}(x_0, x_1, \ldots, x_l)] / z_0, z_1, \ldots, z_l \)

\[ \sim_L d_{L+1}(z_0, z_1, \ldots, z_l), [\sigma_{L+1}(x_0, x_1, \ldots, x_l)] / z_0, z_1, \ldots, z_l \]

Since \( z_i \) are fresh, (16) can be rewritten as:

(16) \( dpool \vdash d_{L+1}(z_0, z_1, \ldots, z_l), [\sigma_{L+1}(x_0, x_1, \ldots, x_l)] / z_0, z_1, \ldots, z_l \)

\[ \sim_L d_{L+1}(z_0, z_1, \ldots, z_l), [\sigma_{L+1}(x_0, x_1, \ldots, x_l)] / z_0, z_1, \ldots, z_l \]

Combining (14) and (17), we have:

(18) \( dpool \vdash d_0(z_0), [\sigma_0(z_0)] / x_0 \)

\[ \sim_0 d_1(z_0), [\sigma_1(z_0)] / x_0 \]

\[ \ldots \]

\[ \sim_0 d_{L+1}(z_0, z_1, \ldots, z_l), [\sigma_{L+1}(x_0, x_1, \ldots, x_l)] / z_0, z_1, \ldots, z_l \]

\[ \ldots \]

\[ \sim_0 d_{L+1}(z_0, z_1, \ldots, z_l), [\sigma_{L+1}(x_0, x_1, \ldots, x_l)] / z_0, z_1, \ldots, z_l \]

The conclusion holds \( \square \)

**Lemma 14 (Zero Unrolling).**

If \((c(x'), d(x'_d); p(x')) \in dpool(p)\) and \( \vdash c(x') \sigma \) where \( \text{dom}(\sigma) = x_c \), then

either \( \exists \bar{\text{rec}} (\bar{x}_d) \subseteq \text{dom}(\bar{\sigma}) \)

and \( \forall \bar{\sigma} \geq \sigma \) where \( x_d \subseteq \text{dom}(\bar{\sigma}) \),

\[ \exists B \text{ s.t. } \sigma \vdash d(x') \bar{\sigma} : p(x') \bar{\sigma} \]

or \( \exists \bar{\text{rec}} (s(x'_d)) \text{ s.t. } \text{rec}(s(x'_d)) \in d(x'_d) \), s.t.

\[ \exists B \text{ s.t. } \forall \bar{x}_d, s(x'_d) \in d(x'_d), \]

\[ \exists d_1(x'_d), k_s, \sigma_s \text{ s.t. } \]

\[ dpool \vdash (\text{rec}, s(x'_d)), \sigma \bar{x}_d \sim \bar{x}_d \sim d_1(x'_d), \sigma_s \]

and \( \exists \sigma_s \geq \sigma_s \text{ s.t. } \sigma \vdash d_1(x'_d) \sigma_s : s(x'_d) \sigma_s \)

implies

\[ \exists \bar{d}'(x'_d), L, \sigma' \text{ s.t. } \]

\[ dpool \vdash d(x'_d), \sigma' \sim d'(x'_d), \sigma' \]

and \( \exists \sigma' \geq \sigma' \text{ s.t. } \sigma' \vdash d'(x'_d) \sigma' : p(x') \sigma' \)

**Proof.**

**By induction on the structure of** \( d(x'_d) \).

By assumption,

(A) \((c(x'), d(x'_d); p(x')) \in dpool(p)\)

and \( \vdash c(x') \sigma \).

| Base Case: \( d(x'_d) = (BT, p(x')) \) |

By inspection,

(b1) \((BT, p(x'))\) does not contain any \((\text{rec}, s(x'_d))\).

(b2) Pick any \( \bar{\sigma} \geq \sigma \text{ s.t. } x_d \subseteq \text{dom}(\bar{\sigma}) \).

Using (b2) and the fact that \( p \) is a base tuple,

(b3) Define \( B = \{ p(x') \bar{\sigma} \} \)

By (b3),

(b4) \( \sigma \vdash (BT, p(x')) \bar{\sigma} : p(x') \bar{\sigma} \)

By (b1), (b2), (b3), and (b4),

The conclusion holds
Inductive Case: \( d(\vec{x}_d) = (rID, p(\vec{x}), (D_1(X_{d1}^{\ast})):q_1(\vec{x}_1)):\ldots:(D_n(X_{dn}^{\ast}):q_n(\vec{x}_n))::\text{nil}) \)

There are two cases to consider:
(N) \( d(\vec{x}_d) \) does not contain any \( (\text{rec}, \_\_\_) \)
(R) \( \exists (\text{rec}, s(\vec{x}_s)) \) s.t. \( (\text{rec}, s(\vec{x}_s)) \in d(\vec{x}_d) \)

(N) \( d(\vec{x}_d) \) does not contain any \( (\text{rec}, \_\_\_) \)

(N1) Pick any \( \bar{\sigma} \geq \sigma \) where \( \vec{x}_d \subseteq \text{dom}(\bar{\sigma}) \).

Using (N1),
(N2) Define \( B = \{ b \text{ is a base tuple} | b \in d(\vec{x}_d)\bar{\sigma} \} \)

By (N2),
(N3) \( \text{prog}, B \vdash (\text{BT}, p(\vec{x}))|\bar{\sigma}; p(\vec{x})\bar{\sigma} \)

By (N1), (N2), and (N3)

The conclusion holds
(R) \( \exists (\text{rec}, s(\vec{x}_s)) \) s.t. \( d(\vec{x}_d) \)

The rule that derives \( p \) is
\( rID \ p(\vec{u}) : = q(u_1), \ldots, q_n(u_n), c_{pr}(\vec{u}, u_1, \ldots, u_n) \)

There are two cases to consider:
Case A: \( \forall i \in \{1, 2, \ldots, n\}, q_i \) is on a cycle in \( \mathcal{G} \)
Case B: \( \exists i \in \{1, 2, \ldots, n\} \) s.t. \( q_i \) is not on a cycle in \( \mathcal{G} \)

Case A: \( \forall i \in \{1, 2, \ldots, n\}, q_i \) is on a cycle in \( \mathcal{G} \)

By Function \( \text{GENDPool} \),
\( \forall i \in \{1, 2, \ldots, n\}, \)
(A1) \( (c_{\text{inc}}(\vec{z}), \Delta_{\text{inc}}) \in dpool(q_i) \)
\( \vec{z} \) are fresh variables corresponding to \( \vec{x}_i \),
and \( D_i(X_{di}^{\ast}) = (\text{rec}, q_i(\vec{x}_i)) \), thus \( X_{di}^{\ast} = \vec{x}_i \)

Pick any \( B \).

Assume the following:
(A2) \( \exists (\text{rec}, s(\vec{x}_s)) \in (rID, p(\vec{x}), (D_1(X_{d1}^{\ast}):q_1(\vec{x}_1)):\ldots:(D_n(X_{dn}^{\ast}):q_n(\vec{x}_n))::\text{nil}), \)

By (A1) and (A2), we apply the Induction Hypothesis to obtain:
(A3) \( \forall i \in \{1, 2, \ldots, n\}, \)

By (A3), we have
(A4) \( \forall i \in \{1, 2, \ldots, n\}, dpool \vdash (\text{rec}, q_i(\vec{x}_i)), \sigma|_{\vec{x}_i} \mapsto_{k_i} d_i(\vec{x}_d^{\ast}), \sigma_i \)

Since \( \vec{x}_d^{\ast} \), \( \ldots, \vec{x}_d^{\ast} \) are fresh,
(A5) \( \forall i \in \{1, 2, \ldots, n\}, \sigma|_{\vec{r}x^{\ast}_{d1}, \ldots, \vec{r}x^{\ast}_{dn}}, \vec{x}_d^{\ast} \cup \sigma_i \) is well-defined.

By (A5), we apply Weakening (Lemma 11) to (A4) to obtain
(A6) \( \forall i \in \{1, 2, \ldots, n\}, dpool \vdash (\text{rec}, q_i(\vec{x}_i)), \sigma \mapsto_{k_i} d_i(\vec{x}_d^{\ast}), \sigma|_{\vec{r}x^{\ast}_{d1}, \ldots, \vec{r}x^{\ast}_{dn}} \cup \sigma_i \)

In (A6), \( \forall i \in \{1, 2, \ldots, n\} \) s.t. \( k_i < \max(k_1, \ldots, k_n) \), apply Rule \( \text{WAND} \) to obtain:
(A7) \( \forall i \in \{1, 2, \ldots, n\}, dpool \vdash (\text{rec}, q_i(\vec{x}_i)), \sigma \mapsto_{\max(k_1, \ldots, k_n)} d_i(\vec{x}_d^{\ast}), \sigma|_{\vec{r}x^{\ast}_{d1}, \ldots, \vec{r}x^{\ast}_{dn}} \cup \sigma_i \)

By (A7), apply Unrolling Subderivations (Lemma 9) to obtain
(A8) \( dpool \vdash d(\vec{x}_d), \sigma \mapsto_{\max(k_1, \ldots, k_n)+1} (rID, p(\vec{x}), (d_1(\vec{x}_{d1}^{\ast}):q_1(\vec{x}_1)):\ldots:(d_n(\vec{x}_{dn}^{\ast}):q_n(\vec{x}_n))::\text{nil}), \sigma|_{\vec{x}} \cup \bigcup_{i=1}^n \sigma_i \)

Using (A7) and (A8),
(A9) Define \( \sigma = \sigma|_{\vec{x}} \cup \bigcup_{i=1}^n \sigma_i \).
Since $\vec{x}, x_{d1}, \ldots, x_{dn}$ are fresh, thus $\sigma$ is well-defined.
Then $\sigma = (\sigma|x \cup \bigcup_{i=1}^{n} \sigma_i) \geq (\sigma|x \cup \bigcup_{i=1}^{n} \sigma_i) \geq \sigma$

By (A) and (A6),
(A10) $= c_{pr}(\vec{x}, x_{d1}, \ldots, x_{dn}) \sigma$

Using (A4) and (A7), we can apply $rID$ to obtain:
(A11) $prog, B \vdash (rID, \pi(x)), (d_{1}(x_{d1}):q_{1}(x_{i})): \ldots: (d_{1}(x_{d1}):q_{1}(x_{i})):\pi(x_{i})|\sigma|p(\vec{x}) \sigma$

Case B: $\exists i \in \{1, 2, \ldots, n\}$ s.t. $q_i$ is not on a cycle in $G$

By assumption,
Exists an index set $\Lambda = \{\lambda_1, \ldots, \lambda_n\}$ with properties:
$\Lambda \neq \emptyset$
$1 \leq \lambda_i \leq n$
and $q_{\lambda_i}$ is not on a cycle in $G$

By Function GENDPOOL,
(B1) $\forall \lambda_i \in \Lambda,$
\[ \exists (c_{\lambda_i}(z_{d\lambda_i}), d_{\lambda_i}(z_{d\lambda_i}):q_{\lambda_i}(x_{i}')), (d_{1}(x_{d1}):q_{1}(x_{i})): \ldots: (d_{1}(x_{d1}):q_{1}(x_{i})):\pi(x_{i})|\sigma|p(\vec{x}) \sigma \]

(B2) $\forall j \in \{1, 2, \ldots, n\} \setminus \Lambda,$
\[ (c_{\lambda_{\nabla}(\vec{x}_i)}, \Delta_{q_{j}}) \in dpool(q_{j}), \vec{z}_j \text{ are fresh variables corresponding to } X_{d\lambda_i} \]
\[ D_{j}(\vec{X}_{d\lambda_j}) = (\text{rec. } q_{j}(\vec{z}_i)) \]

By (A), (B1), (B2),
(B3) $c(\vec{x}_i) = c_{pr}(\vec{x}, x_{d1}, \ldots, x_{dn}) \land \bigwedge_{i=1}^{n} C_{i}(X_{d\lambda_i})$
\[ \begin{cases} 
\text{either } i \in \Lambda, \text{ thus } q_i \text{ is not on a cycle in } G \text{ and } C_{i}(X_{d\lambda_i}) = c_{i}(x_{i}'), \\
\text{or } i \in \{1, 2, \ldots, n\} \setminus \Lambda, \text{ thus } q_i \text{ is on a cycle in } G \text{ and } C_{i}(X_{d\lambda_i}) = c_{i\text{rec}.}(x_{i}') \end{cases} \]

By (IB3),
(IB5) $\forall \lambda_i \in \Lambda, \exists c_{\lambda_i}(z_{d\lambda_i})|\sigma_{\lambda_i}, \text{ where } \sigma_{\lambda_i} = [\sigma(x_{d\lambda_i})/z_{d\lambda_i}]$

Using (IB1) and (IB5), apply the Inductive Hypothesis to obtain:
(IB6) $\forall \lambda_i \in \Lambda,$
\[ \begin{cases} 
\text{either } d_{\lambda_i}(z_{d\lambda_i}) \text{ does not contain any } (\text{rec. } x) \\
\text{and } \forall \sigma_{\lambda_i} \geq \sigma_{\lambda_i}, \text{ where } x_{d\lambda_i} \subseteq \text{dom}(\sigma_{\lambda_i}), \\
\exists B_{\lambda_i} \text{ s.t. } \text{prog. } B_{\lambda_i} \vdash d_{\lambda_i}(z_{d\lambda_i})|\sigma_{\lambda_i}, q_{\lambda_i}(x_{i}'), \sigma_{\lambda_i}, \text{ or } \exists (\text{rec. } s(z_{i}')) \text{ s.t. } (\text{rec. } s(z_{i}')) \in d_{\lambda_i}(z_{d\lambda_i}), \\
\text{and } \exists B_{\lambda_i} \text{ s.t. } \\
\forall (\text{rec. } s(z_{i}')) \in d_{\lambda_i}(z_{d\lambda_i}), \\
\exists (d_{\lambda_i}(z_{d\lambda_i}), \sigma_{\lambda_i}) \text{ s.t. } \\
dpool : (\text{rec. } s(z_{i}')), \sigma_{\lambda_i} \mid z_{i}' \rightarrow_{\lambda_i} d_{\lambda_i}(z_{d\lambda_i}), \sigma_{\lambda_i}, \\
\text{and } \exists \sigma_{s} \geq \sigma_{s} \text{ s.t. } \text{prog. } B_{\lambda_i} \vdash d_{\lambda_i}(z_{d\lambda_i})|\sigma_{s}:s(z_{i}'), \sigma_{s} \\
\text{implies } \\
\exists d'_{\lambda_i}(z_{d\lambda_i}), L_{\lambda_i}, \sigma'_{\lambda_i} \text{ s.t. } \\
dpool : d'_{\lambda_i}(z_{d\lambda_i}), \sigma'_{\lambda_i} \rightarrow_{\lambda_i} d'_{\lambda_i}(z_{d\lambda_i}), \sigma'_{\lambda_i}, \\
\text{and } \exists \sigma'_{s} \geq \sigma'_{s} \text{ s.t. } \text{prog. } B_{\lambda_i} \vdash d'_{\lambda_i}(z_{d\lambda_i})|\sigma'_{s}:s(z_{i}'), \sigma'_{s} \\
\end{cases} \]

By (IB6),
\[ \forall \lambda_i \in \Lambda, \text{ if } d_{\lambda_i}(z_{d\lambda_i}) \text{ does not contain any } (\text{rec. } x), \]
Pick $\sigma_{s} \geq \sigma_{s} \text{ s.t. } z_{d\lambda_i} = \text{dom}(\sigma_{\lambda_i})$

By the above, and using (IB6),
(IB7) $\forall \lambda_i \in \Lambda, \text{ if } d_{\lambda_i}(z_{d\lambda_i}) \text{ does not contain any } (\text{rec. } x),$
\[ \exists B'_{\lambda_i} \text{ s.t. } \text{prog. } B'_{\lambda_i} \vdash d_{\lambda_i}(z_{d\lambda_i})|\sigma_{\lambda_i}, q_{\lambda_i}(x_{i}'), \sigma_{\lambda_i}, \\
\text{and } \exists \sigma'_{s} \geq \sigma'_{s} \text{ s.t. } \text{prog. } B_{\lambda_i} \vdash d'_{\lambda_i}(z_{d\lambda_i})|\sigma'_{s}:q_{\lambda_i}(x_{i}'), \sigma'_{s} \\
\]

By (IB6) and (IB7),
(IB8) Pick $B = \bigcup_{\lambda_i \in \Lambda} B_{\lambda_i}$
\[ \begin{cases} 
\text{Either } d_{\lambda_i}(z_{d\lambda_i}) \text{ does not contain any } (\text{rec. } x) \text{ and } B_{\lambda_i} = B'_{\lambda_i}, \text{ as in (IB7), } \\
\text{or } \exists (\text{rec. } s(z_{i}')) \text{ s.t. } (\text{rec. } s(z_{i}')) \in d_{\lambda_i}(z_{d\lambda_i}), \text{ and } B_{\lambda_i} = B'_{\lambda_i}, \text{ as in the second case in (IB6). } \\
\end{cases} \]

Using (IB8), assume the following:
By the above and using (IB9), we have:

\[ \exists d_i(\vec{x}_i), k_s, \sigma_s \text{ s.t. } \]
\[ d_{pool} \vdash (\text{rec}, s(\vec{x}_i)), \sigma_{|\vec{x}_i|} \mapsto_{k_s} d_i(\vec{x}_i), \sigma_s \]
\[ \text{and } \exists \sigma_s \geq \sigma_s \text{ s.t. } \text{prog}, B \models d_i(\vec{x}_i)\sigma_s \]

There are three cases to consider:

Case I: \( i \in \Lambda, \) thus \( q_i \) is not on a cycle in \( G \) and \( d_i(z_{\vec{a}_i}) \) does not contain any \( (rec, \_). \)

Case II: \( i \in \Lambda, \) thus \( q_i \) is not on a cycle in \( G \) and \( \exists (\text{rec}, q_i(\vec{x}_i)) \) s.t. \((\text{rec}, q_i(\vec{x}_i)) \in d_i(z_{\vec{a}_i}) \)

Case III: \( i \in \{1, 2, \ldots, n\} \) and \( q_i \) is on a cycle in \( G. \)

Case I: \( i \in \Lambda, \) thus \( q_i \) is not on a cycle in \( G \) and \( d_i(z_{\vec{a}_i}) \) does not contain any \( (rec, \_). \)

By (IB7) and (IB8),
\[ \forall \lambda_i \in \Lambda, \text{ if } d_{\lambda_i}(z_{\vec{a}_i}) \text{ does not contain any } (rec, \_), \]
\[ d_{pool} \vdash d_{\lambda_i}(z_{\vec{a}_i}), \sigma_{\lambda_i} \mapsto_{L_{\lambda_i}} d_{\lambda_i}'(z_{\vec{a}_i}'), \sigma_{\lambda_i}' \]
\[ \text{and } \exists \sigma_{\lambda_i}' \geq \sigma_{\lambda_i} \text{ s.t. } \text{prog}, B_{\lambda_i} \models d_{\lambda_i}'(z_{\vec{a}_i}'), \sigma_{\lambda_i}' \sigma_{\lambda_i} \sigma_{\lambda_i} \]
\[ \text{where } L_{\lambda_i} = 0, d_{\lambda_i}'(z_{\vec{a}_i}) = d_{\lambda_i}(z_{\vec{a}_i}), \text{ and } \sigma_{\lambda_i}' = \sigma_{\lambda_i} \]

By (IB1),
\[ \forall \lambda_i \in \Lambda, z_{\vec{a}_i} \text{ are fresh variables for } x_{\vec{a}_i} \]

By (IB8),
\[ \forall \lambda_i \in \Lambda, \mathcal{B}_{\lambda_i} \subseteq B \]

By the above, applying the substitution \([x_{\vec{a}_i}/z_{\vec{a}_i}]\) and using Substitution (Lemma 13), we have

(IB10) \( \forall \lambda_i \in \Lambda, \text{ if } d_{\lambda_i}(z_{\vec{a}_i}) \text{ does not contain any } (rec, \_), \)
\[ d_{pool} \vdash d_{\lambda_i}(x_{\vec{a}_i}), \sigma_{|x_{\vec{a}_i}|} \mapsto_{L_{\lambda_i}} d_{\lambda_i}'(x_{\vec{a}_i}'), [\sigma_{\lambda_i}'(z_{\vec{a}_i}')] / [x_{\vec{a}_i}] \]
\[ \text{and } \forall \lambda_i \geq \lambda_i \text{ s.t. } \text{prog}, B \models d_{\lambda_i}'(x_{\vec{a}_i}'), [\sigma_{\lambda_i}'(z_{\vec{a}_i}')] / [x_{\vec{a}_i}'] \]

Case II: \( i \in \Lambda, \) thus \( q_i \) is not on a cycle in \( G \) and \( \exists (\text{rec}, q_i(\vec{x}_i)) \) s.t. \((\text{rec}, q_i(\vec{x}_i)) \in d_i(z_{\vec{a}_i}) \)

By construction,
\[ \forall \lambda_i \in \Lambda, (\text{rec}, s(\vec{x}_i)) \in d_{\lambda_i}(x_{\vec{a}_i}) \text{ implies } (\text{rec}, s(\vec{x}_i)) \in d(x_{\vec{a}_i}) \]

By (IB1),
\[ \forall \lambda_i \in \Lambda, z_{\vec{a}_i} \text{ are fresh variables for } x_{\vec{a}_i} \]

By the above and using (IB9), we have:
\[ \forall \lambda_i \in \Lambda, \text{ if } \exists \text{ (rec, s(\vec{x}_i)) \text{ s.t. } (rec, s(\vec{x}_i)) \in d_{\lambda_i}(x_{\vec{a}_i}), \}
\[ \forall (\text{rec, s(\vec{x}_i)}) \in d_{\lambda_i}(x_{\vec{a}_i}), \]
\[ d_{pool} \vdash (\text{rec, s(\vec{x}_i)}), \sigma_{|\vec{x}_i|} \mapsto_{k_s} d_{\lambda_i}(x_{\vec{a}_i}), \sigma_s \]
\[ \text{and } \text{prog}, B \models d_i(x_{\vec{a}_i})\sigma_s \]

By the above, we can apply the substitution \([x_{\vec{a}_i}/z_{\vec{a}_i}]\) and using Substitution (Lemma 13) to obtain:
\[ \forall \lambda_i \in \Lambda, \text{ if } \exists (\text{rec, s(\vec{x}_i)) \text{ s.t. } (rec, s(\vec{x}_i)) \in d_{\lambda_i}(x_{\vec{a}_i}), \]
\[ \forall (\text{rec, s(\vec{x}_i)}) \in d_{\lambda_i}(x_{\vec{a}_i}), \]
\[ d_{pool} \vdash (\text{rec, s(\vec{x}_i)}), [\sigma(\vec{x}_i)/\vec{z}_i] \mapsto_{k_s} d_{\lambda_i}(x_{\vec{a}_i}), [\sigma_s(x_{\vec{a}_i})/\vec{z}_i] \]
\[ \text{and } \forall \lambda_i \geq \lambda_i \text{ s.t. } \text{prog}, B \models d_{\lambda_i}(x_{\vec{a}_i})\sigma_s(x_{\vec{a}_i}) / [x_{\vec{a}_i}] \]

Using the above, apply (IB6) to obtain:

(IB11) \( \forall \lambda_i \in \Lambda, \text{ if } \exists (\text{rec, s(\vec{x}_i)}) \text{ s.t. } (rec, s(\vec{x}_i)) \in d_{\lambda_i}(x_{\vec{a}_i}), \)
\[ \exists d_{\lambda_i}(z_{\vec{a}_i}'), L_{\lambda_i}, \sigma_{\lambda_i} \text{ s.t. } \]
\[ d_{pool} \vdash d_{\lambda_i}(z_{\vec{a}_i}'), \sigma_{\lambda_i} \mapsto_{L_{\lambda_i}} d_{\lambda_i}'(z_{\vec{a}_i}'), \sigma_{\lambda_i}' \]
\[ \text{and } \exists \sigma_{\lambda_i}' \geq \sigma_{\lambda_i} \text{ s.t. } \text{prog}, B \models d_{\lambda_i}'(z_{\vec{a}_i}'), \sigma_{\lambda_i}' q_{\lambda_i}(z_{\vec{a}_i}'), \sigma_{\lambda_i}' \]

Apply the substitution \([x_{\vec{a}_i}/z_{\vec{a}_i}]\) and use Substitution (Lemma 13) to (IB11) to obtain:

(IB12) \( \forall \lambda_i \in \Lambda, \text{ if } \exists (\text{rec, s(\vec{x}_i)}) \text{ s.t. } (rec, s(\vec{x}_i)) \in d_{\lambda_i}(x_{\vec{a}_i}), \)
\[ d_{pool} \vdash d_{\lambda_i}(x_{\vec{a}_i}), \sigma_{|\vec{x}_i|} \mapsto_{L_{\lambda_i}} d_{\lambda_i}'(x_{\vec{a}_i}'), [\sigma_{\lambda_i}'(z_{\vec{a}_i}), \sigma_{\lambda_i}' \]
\[ \text{and } \forall \lambda_i \geq \lambda_i \text{ s.t. } \text{prog}, B \models d_{\lambda_i}(x_{\vec{a}_i})\sigma_{\lambda_i}' q_{\lambda_i}(x_{\vec{a}_i}) / [x_{\vec{a}_i}'] \]

Case III: \( i \in \{1, 2, \ldots, n\} \) and \( q_i \) is on a cycle in \( G. \)

By the assumptions made in (IB10),

(IB13) \( \forall j \in \{1, 2, \ldots, n\} \setminus \Lambda, \)
\[ \exists d_j(x_{\vec{a}_j}), L_j, \sigma_j \text{ s.t. } \]
\[ d_{pool} \vdash (\text{rec, q_j(\vec{x}_i)}), \sigma_{|\vec{x}_i|} \mapsto_{L_j} d_j(x_{\vec{a}_j}'), \sigma_j \]
\[ \text{and } \exists \sigma_j \geq \sigma_j \text{ s.t. } \text{prog}, B \models d_j(x_{\vec{a}_j})\sigma_j q_j(x_{\vec{a}_j}) / [x_{\vec{a}_j}] \]

By Sigma Extension (Lemma 6),
\[ \text{dom}(\sigma_j) \subseteq \vec{a}_j \]

Define \( L_\Lambda = \max_{\lambda_i \in \Lambda} \lambda_i. \) Since \( \Lambda \neq \emptyset, L_\Lambda \) is a defined number.

Define \( L_{\text{rec}} = \max_{j \in \{1, 2, \ldots, n\} \setminus \Lambda} L_j. \) If \( \{1, 2, \ldots, n\} \setminus \Lambda = \emptyset, \) set \( L_{\text{rec}} = 0. \)

Set \( \ell = \max(L_\Lambda, L_{\text{rec}}). \)

By (IB11) and (IB13),
∀λi ∈ Λ. dpool ⊢ dλi(x, ⃗ dλi), σ|x, ⃗ dλi → Lλi, d′λi(x, ⃗ dλi ′), [σ′λi/ x, ⃗ dλi]  

Since x, ⃗ d1, ..., ⃗ dn are fresh,  

∀j ∈ {1, 2, ..., n} \ Λ. σ|x, ⃗ x1,..., ⃗ xj−1, ⃗ xj+1,..., ⃗ x \ ∈ σ′ is well-defined.  

By the above, we can apply Weakening (Lemma 11) to the above to obtain:  

∀λi ∈ Λ. dpool ⊢ dλi(x, ⃗ dλi), σ → Lλi, d′λi(x, ⃗ dλi ′), [σ′λi/ x, ⃗ dλi]  

Using Rule Wkind wherever Lλi < ℓ.  

(IB14) ∀λi ∈ Λ. dpool ⊢ dλi(x, ⃗ dλi), σ →t d′λi(x, ⃗ dλi ′), [σ|x, ⃗ x1,..., ⃗ xj−1, ⃗ xj+1,..., ⃗ x ⊨ [σ′λi/ x, ⃗ dλi].  

Since x, ⃗ d1, ..., ⃗ dn are fresh,  

∀j ∈ {1, 2, ..., n}\ Λ. σ|x, ⃗ x1,..., ⃗ xj−1, ⃗ xj+1,..., ⃗ x \ ∈ σ′ is well-defined.  

By the above, we can apply Weakening (Lemma 11) to (IB14) to obtain:  

∀j ∈ {1, 2, ..., n}\ Λ. dpool ⊢ (rec, qj(⃗ x)), σ →t d′j(x, ⃗ x ′ ′), σ|x, ⃗ x1,..., ⃗ xj−1, ⃗ xj+1,..., ⃗ x ⊨ [σ′j/ ⃗ x].  

Using Rule Wkind wherever Lλj < ℓ.  

(IB15) ∀j ∈ {1, 2, ..., n}\ Λ. dpool ⊢ (rec, qj(⃗ x)), σ →t d′j(x, ⃗ x ′ ′), σ|x, ⃗ x1,..., ⃗ xj−1, ⃗ xj+1,..., ⃗ x ⊨ [σ′j/ ⃗ x].  

By (IB13) and (IB14), apply Unrolling Subderivations (Lemma 9) to obtain:  

(IB16) dpool ⊢ (rID, p(x), (D1(⃗ x1); q1(⃗ x1))(::, ..., (Dn(⃗ xn); qn(⃗ xn))(::nil), σ →t+1 (rID, p(x), (d′(x, ⃗ x1); q1(⃗ x1))(::, ..., (d′n(⃗ xn); qn(⃗ xn))(::nil)), σ|x, ⃗ x1,..., ⃗ xj−1, ⃗ xj+1,..., ⃗ x, ⊨ [σ′j/ ⃗ x]).  

Pick σ = σ|x, ⃗ x1,..., ⃗ xj−1, ⃗ xj+1,..., ⃗ x ⊨ [σ′j/ ⃗ x].  

Since ⃗ x, ⃗ x1,..., ⃗ xn are fresh, σ is well-defined.  

By (IB11) and (IB13),  

σ ≥ σ|x, ⃗ x1,..., ⃗ xj−1, ⃗ xj+1,..., ⃗ x ⊨ [σ′j/ ⃗ x].  

By (A) and (IB4),  

(IB17) c_{rec}(⃗ x, ⃗ x1,..., ⃗ xn)σ.  

By (IB11), (IB13), and (IB16), we can apply rID to obtain:  

(IB18) prog, B ⊢ (rID, p(x), (d′(x, ⃗ x1); q1(⃗ x1))(::, ..., (d′n(⃗ xn); qn(⃗ xn))(::nil))σ; p(x)σ.  

B.2.2 Proof of Correctness Of Derivation Pool Construction  

Theorem 5 (Correctness Of Derivation Pool Construction (Recursive)).  

DGRAPH(prog) = G, and GENDPOOL(G, A) = dpool  

1. If prog, B ⊢ d:p(⃗ t), then  

either p is not on a cycle in G and  

∃c(⃗ x), d(⃗ x′): p(⃗ x) ∈ dpool(p),  

∃σ′, σ, t.s.t. c(⃗ x)|σd| ⃗ x,  

∃d′(⃗ x′): p(⃗ x′),  

∃σ, t.s.t. dpool ⊢ d′(⃗ x′), σ|d′(⃗ x′) →t d′(⃗ x′) ′ ′, σ′c  

and d = d′(⃗ x′)σ ′ ′,  

or p is on a cycle in G and  

∃c(⃗ x), Δp, p ∈ dpool(p),  

∃σ′, σ, t.s.t. c(⃗ x)|σd| ⃗ x,  

∃d′(⃗ x′): p(⃗ x′),  

∃σ, t.s.t. dpool ⊢ (rec, p(⃗ x)), σ ′ ′ |d′(⃗ x′) →t d′(⃗ x′) ′ ′, σc  

and d = d′(⃗ x′)σ ′ ′.  

2. ∀ℓ ∈ N,  

(a) If c(⃗ x), d(⃗ x): p(⃗ x) ∈ dpool(p) and c(⃗ x), σ where dom(σ) = ⃗ x, then  

either ∃m ≤ ℓ, d′(⃗ x′), σ′ s.t.  

and dpool ⊢ d(⃗ x), σ →m d′(⃗ x′), σ′, and d′(⃗ x′) does not contain any (rec, _)  

and ∀σ′ ≥ σ′ where ⃗ x′ ⊂ dom(σ ′),  

and B, s.t. prog, B ⊢ d′(⃗ x′)σ′; p(⃗ x)σ′,  

or ∃d′(⃗ x′), σ′ s.t.  

dpool ⊢ d(⃗ x), σ →t d′(⃗ x′), σ′, and  

and ∃B, s.t. (rec, s(⃗ x)) s.t. (rec, s(⃗ x)) ∈ d′(⃗ x′)  

and B, s.t.  

∀(rec, s(⃗ x)) ∈ d′(⃗ x′),  

∅d(⃗ x), s, σ, t.s.t.
dpool ⊨ (rec, s(x′_s)), σ′|x_s →_k_s d_s(x_{ds}), σ_s
and ∃σ_s ≥ σ_s s.t. prog, B ⊨ d_s(x_{ds})σ_s:s(x′_s)σ_s
implies
∃d′′(x_{d′′}), L, σ'' s.t.
dpool ⊨ d(x_{ds}), σ →_0 d(x_{d′}), σ →_1 ... →_ℓ d′(x_{d′}), σ' →_{ℓ+1} ... →_{ℓ+L} d''(x_{d''}), σ''
and ∃σ'' ≥ σ'' s.t. prog, B ⊨ d''(x_{d''})σ'' :p(x)σ''
(b) If (c_{rec,p}(x), Δ_p) ∈ dpool(p) and ⊨ c_{rec,p}(x)σ where dom(σ) = x, then
either ∃m ≤ ℓ, d′(x_{d′}), σ s.t.
and dpool ⊨ (rec, p(x)), σ →_m d′(x′_d), σ',
and d′ does not contain any (rec, p),
and ∀σ' ≥ σ where x_{d′} ⊇ dom(σ'),
∃B s.t. prog, B ⊨ d′(x′_d)σ':p(x)σ',
or ∃d′(x′_d), σ' s.t.
dpool ⊨ (rec, p(x)), σ →_t d′(x′_d), σ'
and ∃(rec, s(x′_s)) s.t. (rec, s(x′_s)) ∈ d′(x′_d)
and ∃B s.t.
∀(rec, s(x′_s)) ∈ d′(x′_d),
∃d_s(x_{ds}), k_s, σ_s s.t.
dpool ⊨ (rec, s(x′_s)), σ′|x_s →_k_s d_s(x_{ds}), σ_s
and ∃σ_s ≥ σ s.t. prog, B ⊨ d_s(x_{ds})σ_s:s(x′_s)σ_s
implies
∃d′′(x_{d′′}), L, σ'' s.t.
dpool ⊨ (rec, p(x)), σ →_0 (rec, p(x)), σ →_1 ... →_ℓ d′(x_{d′}), σ' →_{ℓ+1} ... →_{ℓ+L} d''(x_{d''}), σ''
and ∃σ'' ≥ σ'' s.t. prog, B ⊨ d''(x_{d''})σ'' :p(x)σ''
1. Proof by induction of the structure of $d$

By assumption,

\((*)\) \text{prog}, B \vdash d : p(\vec{t})

---

**Base Case: $d = (BT, p(\vec{t}))$**

Since $p$ is a base tuple,

- (B1) $p$ is not on a cycle in $\mathcal{G}$
- By (B1) and GENDPool,
  - (B2) $\langle T, (BT, p(\vec{x})) \rangle \in dpool(p)$

- (B3) Define $\sigma = [\vec{t} / \vec{x}]$.

By definition of "\$\top\$",

\[ \forall \sigma, B \vdash \top \sigma \]

Pick $\bar{\sigma} = \sigma$. Then by the above, we have:

- (B4) $B \vdash \top \bar{\sigma}$

By \text{BASE},

\[ \forall \sigma, dpool \vdash (BT, p(\vec{x})), \bar{\sigma} \leadsto_0 (BT, p(\vec{x})), \bar{\sigma} \]

Pick $\bar{\sigma} = \sigma$. The by the above, we have:

- (B5) $dpool \vdash (BT, p(\vec{x})), \sigma \leadsto_0 (BT, p(\vec{x})), \sigma$

By construction,

- (B6) $d = (BT, p(\vec{x})) \bar{\sigma}$

---

**Inductive Case: $d = (rID, p(\vec{u}), (d_1 : q_1(\vec{t}_1)) : \ldots : (d_n : q_n(\vec{t}_n)) : \text{nil})$**

We denote the rule used for deriving $p$ in $d$ as

\[ rID \vdash p(\vec{u}) : = q_1(\vec{u}_1), \ldots , q_n(\vec{u}_n), c_{pr}(\vec{u}, \vec{u}_1, \ldots , \vec{u}_n) \]

By \((*)\),

- (I1) $\vdash c_{pr}(\vec{t}, \vec{t}_1, \ldots , \vec{t}_n)$
- (I2) $\forall i \in \{1, 2, \ldots , n\}, \text{prog}, B \vdash d_i : q_i(\vec{t}_i)$

By (I2), we apply the Induction Hypothesis to obtain:

- (I3) $\forall i \in \{1, 2, \ldots , n\}$,
  - either $q_i$ is not on a cycle in $\mathcal{G}$ and
    \[ \exists (\bar{c}(\vec{z}_i^{'}) , \bar{d}(\vec{z}_i^{''}), \bar{c}(\vec{z}_i^{''})) \in dpool(p_i), \]
    \[ \exists \sigma_{di}^{''} \text{ s.t. } \bar{c}(\vec{z}_i^{'}) \sigma_{di}^{''} \bar{z}_i^{''}, \]
    \[ \exists \sigma_{ci}^{''} \text{ s.t. } \quad \bar{d}(\vec{z}_i^{''}), \sigma_{ci}^{''} \bar{z}_i^{''}, \]
    \[ \text{dpool } \vdash \bar{d}(\vec{z}_i^{''}), \sigma_{ci}^{''} \bar{z}_i^{''} \]
    \[ \text{and } d_i = d_i^{''}(\vec{z}_i^{''}), \]
  - or $q_i$ is on a cycle in $\mathcal{G}$ and
    \[ \exists (\bar{c}_{rec-pi}(\vec{z}_i^{'}) , \Delta_{pi}) \in dpool(p_i), \]
    \[ \exists \sigma_{di}^{''} \text{ s.t. } \bar{c}_{rec-pi}(\vec{z}_i^{'}) \sigma_{di}^{''} \bar{z}_i^{''}, \]
    \[ \exists \sigma_{ci}^{''} \text{ s.t. } \quad \Delta_{pi} \sigma_{ci}^{''} \bar{z}_i^{''}, \]
    \[ \text{dpool } \vdash \bar{c}_{rec-pi}(\vec{z}_i^{'}) , \sigma_{ci}^{''} \bar{z}_i^{''} \]
    \[ \text{and } d_i = d_i^{''}(\vec{z}_i^{''}). \]

Using (I3),

- (I4) $\forall i \in \{1, 2, \ldots , n\}$, denote $\sigma_{di}^{''} = [\bar{t}_{di}^{''} / \bar{z}_i^{''}]$

By R\text{NREC} and R\text{REC},

- (I5) $\forall i \in \{1, 2, \ldots , n\},$
  - $\bar{z}_i^{''}$ are fresh.
and either \( q_i \) is not on a cycle in \( G \) and \( z_i' \subseteq z_{d_i} \), \( z_{c_i} \subseteq z_{d_i} \), \( z_{d_i} \subseteq z_{d_i}'' \)

or \( q_i \) is on a cycle in \( G \) and \( z_i \subseteq z_{d_i}'' \)

Subcase 1: \( p \) is not on a cycle in \( G \)

By GENDPOOL and GENDRULE.

1. By assumption, 
2. (I1) and (I3),
3. (I4) and (I5),
4. Define substitution \( \sigma \)
5. Using (I4) and (I5), define substitution \( \sigma_p \) as follows:
6. where \( \sigma_p \) is well-defined because \( y_{d_1}'' \) are fresh

By (1), (2), (3),

1. \( \sigma_p \) is substitution well-defined.
2. \( \sigma_p \) is substitution well-defined.

By Substitution (Lemma 13) and (I3),

- \( \forall i \in \{1, 2, \ldots, n\} \), dpool \( \vdash D_i'(Y_{d_i}) \), \( [Z_{d_i}' / Y_{d_i}'] \sigma_{d_1}'', \ldots, \sigma_{d_n}'' \]

Which is equal to

1. (1.4) \( \forall i \in \{1, 2, \ldots, n\} \), dpool \( \vdash D_i'(Y_{d_i}) \), \( \sigma_p \) \( \vdash \) \( \downarrow \)

By (I5),

- \( \forall i \in \{1, 2, \ldots, n\} \), \( \sigma_p \) \( \vdash \) \( \downarrow \)

By the above, we can apply Weakening (Lemma 11) to (1.4) to obtain:

1. (1.5) \( \forall i \in \{1, 2, \ldots, n\} \), dpool \( \vdash D_i'(Y_{d_i}) \), \( \sigma_p \) \( \vdash \) \( \downarrow \)

By applying WKind to (1.5), we have:

1. (1.6) \( \forall i \in \{1, 2, \ldots, n\} \), dpool \( \vdash D_i'(Y_{d_i}) \), \( \sigma_p \) \( \vdash \) \( \downarrow \)

By Unrolling Subderivations (Lemma 9) and (1.6),

1. (1.7) dpool \( \vdash (rID, p(y), (D_i'(Y_{d_i}')):q_i(y))::(D_i'(Y_{d_i}')):q_i(y))::nil), \sigma_p \) \( \vdash \) \( \downarrow \)

By the definition of \( \sigma_p \),

1. (1.8) \( (rID, p(y), (D_i'(Y_{d_i}')):q_i(y))::(D_i'(Y_{d_i}')):q_i(y))::nil) \( \vdash \)

Subcase 2: \( p \) is on a cycle in \( G \)

By GENDRULE.

1. (2.1) \( (c_{\text{rec}}(\bar{x}), \Delta_p) \in \text{dpool}(p) \),

By assumption,  

2. \( \exists j \in \{1, 2, \ldots, m\} \) s.t. \( (c_{\text{rec}}(\bar{x}), \Delta_p) \) \( \in \) \( \text{dpool}(p) \),

where \( \Delta_p = (c_{\text{rec}}(\bar{x}), \Delta_p) \) \( \in \) \( \text{dpool}(p) \)

and \( \Delta_p \) is either \( \Delta_p \) or \( \text{dpool}(p) \)

and \( \Delta_p = (rID, p(\bar{x}), (D_i'(X_{d_i}')):q_i(x_{d_i}))::(D_i'(X_{d_i}')):q_i(x_{d_i}))::nil), p(x_{d_i}))::nil) \)

where \( D_i'(X_{d_i}')) \) is either \( d_i'(x_{d_i}) \) or \( (\text{rec}, q_i(x_{d_i})) \)

and \( X_{d_1}, \ldots, X_{d_n} \) are fresh variables corresponding to \( Z_{d_1}, \ldots, Z_{d_n} \) respectively
To apply \( \text{RRec} \) rule, we generate fresh variables for \( X_{d_1}', \ldots, X_{d_n}' \), which we name \( W_{d_1}', \ldots, W_{d_n}' \).

By (13) and \( \text{RRec} \), we have

\[
(2.4) \ (rID, p(x), (d_1'(w_{d_1}'':q_1(w_1')): \ldots :d_n'(w_{d_n}'':q_n(w_n')):\nil); p(y))
\]

where \( w_{d_1}'', \ldots, w_{d_n}'' \) are fresh variables corresponding to \( z_{d_1}, \ldots, z_{d_n} \).

and \( \forall i \in \{1, 2, \ldots, n\} \), \( W_{d_i} \subseteq \{\tilde{w}_{d_i}''\} \)

Define a substitution \( \sigma_p \) as follows

\[
(2.5) \ \sigma_p = [\bar{t} \bar{x}] \bigcup \bigcup_{i=1}^{n} [\tilde{d}_i'/\tilde{w}_{d_i}']
\]

where \( \sigma_p \) is well-defined because \( w_{d_1}'', \ldots, w_{d_n}'' \) are fresh

By (2.3) and (2.9),

\[
(2.6) \vdash_{\text{corep}} (\bar{x}) \sigma_p | \bar{x}
\]

By Substitution (Lemma 13) and (4),

\[
(2.7) \ \forall i \in \{1, 2, \ldots, n\}, \text{dpool} \vdash D_1' (W_{d_1}'), [Z_{d_i}'/W_{d_i}'] \sigma_{d_i}' \rightarrow [d_i'(w_{d_i}'')] \rightarrow \[z_{d_i}''/w_{d_i}'] \sigma_{c_i}
\]

The above is equal to

\[
(2.8) \ \forall i \in \{1, 2, \ldots, n\}, \text{dpool} \vdash D_1' (W_{d_1}'), \sigma_p | W_{c_i}' \rightarrow [d_i'(w_{d_i}'')] \rightarrow \[z_{d_i}''/w_{d_i}'] \sigma_{c_i}
\]

Since \( \tilde{w}_{c_1}', \ldots, \tilde{w}_{c_1}' \) are fresh,

\[
(2.9) \ \forall i \in \{1, 2, \ldots, n\}, \sigma_p | W_{c_i}' \rightarrow \[z_{d_i}''/w_{d_i}'] \sigma_{c_i}
\]

By applying \( \text{WIND} \) rule on (2.8),

\[
(2.10) \ \text{dpool} \vdash D_1' (W_{d_1}'), \sigma_p | W_{c_1}' \rightarrow \max(d_1, \ldots, d_n) \rightarrow [d_i'(w_{d_i}'')] \rightarrow \[z_{d_i}''/w_{d_i}'] \sigma_{c_i}
\]

By Unrolling Subderivations (Lemma 9) and (2.9),

\[
(2.11) \ \text{dpool} \vdash (rID, p(x), (D_1'(W_{d_1}')): q_1(w_1')): \ldots : (D_n'(W_{d_n}')): q_n(w_n')): \nil), \sigma_p | W_{c_1}' \rightarrow \max(d_1, \ldots, d_n) + 1
\]

Since \( \tilde{w}_{c_1}', \ldots, \tilde{w}_{c_1}' \) are fresh,

\[
(2.12) \ \text{dpool} \vdash (rID, p(x), (D_1'(W_{d_1}')): q_1(w_1')): \ldots : (D_n'(W_{d_n}')): q_n(w_n')): \nil), \sigma_p | W_{c_1}' \rightarrow \max(d_1, \ldots, d_n) + 1
\]

By (11) and (13),

\[
(2.13) \ \text{dpool} \vdash (\text{rec}, p(x), \sigma_p | x) \rightarrow_{\text{rec}} (\text{rec}, p(x), \sigma_p | x)
\]

By applying \( \text{BASE} \) rule, we have

\[
(2.14) \ \text{dpool} \vdash (\text{rec}, p(x), \sigma_p | x) \rightarrow_{\text{rec}} (\text{rec}, p(x), \sigma_p | x)
\]

By (2.1), (2.2), and (2.9), applying \( \text{RRec} \) rule, we have

\[
(2.15) \ \text{dpool} \vdash (\text{rec}, p(x), \sigma_p | x) \rightarrow_{\text{rec}} (\text{rec}, p(x), D_1'(W_{d_1}')): \ldots : D_n'(W_{d_n}')): \nil), \sigma_p | W_{c_1}' \rightarrow \max(d_1, \ldots, d_n) + 1
\]

By the above and (2.10),

\[
(2.16) \ \text{dpool} \vdash (\text{rec}, p(x), \sigma_p | x) \rightarrow_{\text{rec}} (\text{rec}, p(x), D_1'(W_{d_1}')): \ldots : D_n'(W_{d_n}')): \nil), \sigma_p | x \rightarrow \max(d_1, \ldots, d_n) + 1
\]
2. Proof by induction on $\ell$.

**Base Case: $\ell = 0$**

**Subcase (a): $p$ is not on a cycle in $dpool$**

Since $\ell = 0$,

$m \leq \ell$ implies $m = 0$

By (a2) and BASE rule,

$d(x_{\ell}), \sigma \sim_0 d(x_{\ell}), \sigma$

By Zero Unrolling (Lemma 14),

the conclusion holds

**Subcase (b): $p$ is on a cycle in $dpool$**

By assumption,

(b1) $(c(x), \Delta_p) \in dpool(p)$

and $\vdash c(x)\sigma$

By assumption,

(b2) $dpool \vdash (rec, p(\vec{x})), \sigma \sim_0 d'(\vec{x}')$, $\sigma'$

By applying inversion on BASE rule to (b2),

(b3) $d'(\vec{x}') = (rec, p(\vec{x}))$ and $\sigma' = \sigma$

By (b3),

(b4) $dpool \vdash (rec, p(\vec{x})), \sigma \sim_0 (rec, p(\vec{x})), \sigma$

Pick any $B$

Using (b4), assume that

(b5) $\exists d_p(x_{\ell p}), k \geq 1, \sigma_p$ s.t.

$dpool \vdash (rec, p(\vec{x})), \sigma \sim_0 (rec, p(\vec{x})), \sigma \sim_0 \ldots \sim_0 d_p(x_{\ell p}), \sigma_p$

and prog, $B \vdash d_p(x_{\ell p})\sigma_p; p(\vec{x})\sigma_p$

Pick

(b6) $d''(\vec{x}'') = d_p(x_{\ell p}), L = k, \sigma'' = \sigma_p$

By (b5) and (b6),

the conclusion follows

**Inductive Case: $\ell = k + 1$**

**Subcase (N): $p$ is not on a cycle in $G$**

By assumption,

(N1) $(c(x), d(x); p(\vec{x})) \in dpool(p)$

and $\vdash c(x)\sigma$

We need to consider two cases:

Case i: $p$ is a base tuple.

By the algorithm,

$d(x) = (BT, p(\vec{x}))$

$d(x)$ does not contain any $(rec, \_)$, so only the trivial unrolling via WIND is possible.

The proof for this case is straightforward, so we omit the details.

Case ii: $p$ is a derived tuple.

The rule that derives $p$ is

$rID \ p(\vec{u}) :- q_1(\vec{u}_1), \ldots, q_n(\vec{u}_n), c_{pr}(\vec{u}, \vec{u}_1, \ldots, \vec{u}_n)$

Thus, $d(x_{\ell})$ has form:

(N2) $d(x_{\ell}) = (rID, p(x), (D_1(X_{d_1}); q_1(x_1^1)):: \ldots :: (D_n(X_{d_n}); q_n(x_n^1)); nil)$

By GENPOOL,

(N3) $\forall i \in \{1, 2, \ldots, n\}$

either $q_i$ is not a cycle in $G$, thus $\exists (c_i(z_{\ell i}), d_i(z_{\ell i}); q_i(z_{\ell i})) \in dpool(q_i)$, and

$D_i(X_{d_i}) = d_i(z_{\ell i})$
or \( q_i \) is on a cycle in \( G \), thus \( c_{\text{in}}(\vec{z}_i), \Delta_y \) \( \in d_{\text{pool}}(q_i) \) and \( D_i(X_{di}) = (\text{rec}, q_i(x_i)) \).

(N4) \( c(\vec{x}_i) = c_{\text{pr}}(\vec{x}, x_1, \ldots, x_n) \land \bigwedge_{i=1}^n C_i(X_{ci}) \)

either \( q_i \) is not a cycle in \( G \) and \( C_i(X_{ci}) = c(\vec{x}_{ci}) \)

or \( q_i \) is on a cycle in \( G \) and \( C_i(X_{ci}) = c_{\text{in}}(\vec{z}_i) \).

(N5) \( \vec{x}, X_{d1}, \ldots, X_{dn} \) are fresh variables corresponding to \( \vec{z}, Z_{d1}, \ldots, Z_{dn} \), respectively.

By GenDPool, (N3) and (N5),

(N6) \( \forall i \in \{1, 2, \ldots, n\} \),

either \( q_i \) is not on a cycle and \( z_{di} \subseteq z_{di}' \),

or \( q_i \) is on a cycle and \( z_i \subseteq z_{di}' \subseteq z_{di}'' \).

By (N1) and (N5),

(N7) \( \forall i \in \{1, 2, \ldots, n\} \), \( \models C_i(Z_{ci}) \sigma_i \), where \( \sigma_i = [\sigma(X_{ci})/Z_{ci}] \),

and \( \models c_{\text{pr}}(\vec{x}, z_1, \ldots, z_n)[\sigma(\vec{x}, x_1, \ldots, x_n)/\vec{z}, z_1, \ldots, z_n] \).

By (N3) and (N6), we apply the Inductive Hypothesis to obtain:

\( \forall i \in \{1, 2, \ldots, n\} \),

(N8) If \( (c_{\text{in}}(\vec{z}_i), d_i(z_{di}'; q_i(\vec{z}_i))) \in d_{\text{pool}}(q_i) \), then

\( \exists m_i \leq k, d'_i(z_{di}'), \sigma'_i \) s.t.

\( d_{\text{pool}} \models d_i(z_{di}'), \sigma_i \rightarrow m_i, d'_i(z_{di}'), \sigma'_i \)

and \( d'_i \) does not contain any \( (\text{rec}, s(\vec{z}_i)) \)

and \( \forall \sigma'_i \geq \sigma_i \), where \( z_{di}'' \subseteq \text{dom}(\bar{\sigma}_i) \),

\( \models B_i \), s.t. \( \text{prog}, B_i \models d'_i(z_{di}'); \sigma'_i; q_i(\vec{z}) \bar{\sigma}_i \),

or \( \exists d'_i(z_{di}'''), \sigma''_i \) s.t.

\( d_{\text{pool}} \models d_i(z_{di}''), \sigma_i \)

\( \models \bar{\sigma}_i'' \geq \sigma_i '' \) s.t. \( \text{prog}, B_i \models d'_i(z_{di}''); \sigma''_i; q_i(\vec{z}) \bar{\sigma}_i'' \).

(N9) If \( (c_{\text{in}}(\vec{z}_i), \Delta_y) \) \( \in d_{\text{pool}}(q_i) \), then

\( \exists m_i \leq k, d'_i(z_{di}'), \sigma'_i \) s.t.

\( d_{\text{pool}} \models (\text{rec}, q_i(\vec{z}_i)), \sigma_i \rightarrow m_i, d'_i(z_{di}'), \sigma'_i \)

and \( d'_i \) does not contain any \( (\text{rec}, s(\vec{z}_i)) \)

and \( \forall \sigma'_i \geq \sigma_i \), where \( z_{di}'' \subseteq \text{dom}(\bar{\sigma}_i) \),

\( \models B_i \), s.t. \( \text{prog}, B_i \models d'_i(z_{di}'); \sigma'_i; q_i(\vec{z}) \bar{\sigma}_i' \),

or \( \exists d'_i(z_{di}'''), \sigma''_i \) s.t.

\( d_{\text{pool}} \models (\text{rec}, q_i(\vec{z}_i)), \sigma_i \rightarrow k, d'_i(z_{di}'''), \sigma''_i \)

and \( \exists \bar{\sigma}_i'' \geq \sigma_i '' \) s.t. \( \text{prog}, B_i \models d'_i(z_{di}''); \sigma''_i; q_i(\vec{z}) \bar{\sigma}_i'' \).

We need to consider two cases:
(A) ∀i ∈ {1, 2, ..., n}, the first condition in (N7) and (N8) holds.
(B) ∃i ∈ {1, 2, ..., n} s.t. the second condition in (N7) and (N8) holds.

By (N8) and (N9),
(A1) ∀i ∈ {1, 2, ..., n}, dpool ⊢ D_i(Z_{d_i}), σ_i →_m d_i'(z_{d_i}'), σ'_{i},
either q_i is not on a cycle and D_i(Z_{d_i}) = d_i(z_{d_i})
or q_i is on a cycle and D_i(Z_{d_i}) = (rec, q_i(z_i))

By (N8), (N9), (A1),
∀ ∀ i ∈ {1, 2, ..., n}, Z_{d_i} ⊆ z_{d_i}', x_{d_i}' are fresh variables corresponding to z_{d_i}', and x_{d_i} ⊆ x_{d_i}'.
By the above, we can apply the Substitution (Lemma 13) to obtain:
(A2) ∀ i ∈ {1, 2, ..., n}, dpool ⊢ D_i(X_{d_i}), σ|_{X_{d_i}'} →_m d_i'(x_{d_i}'), [σ'_{i}(z_{d_i}')/x_{d_i}']

Since ∀ i ∈ {1, 2, ..., n}, z_{d_i}' are fresh,
∀ i ∈ {1, 2, ..., n}, σ|_{X_{d_i}'} ⊆ X_{d_i}' ⊆ X_{d_i}' ⊆ X_{d_i}' ⊆ X_{d_i}' \sqcup [σ'_{i}(z_{d_i}')/x_{d_i}'] is well-defined

Using the above, we can apply Weakening (Lemma 11) to (A2) to obtain:
(A3) ∀ i ∈ {1, 2, ..., n}, dpool ⊢ D_i(X_{d_i}), σ|_{X_{d_i}'} →_m d_i'(x_{d_i}'), [σ'_{i}(z_{d_i}')/x_{d_i}'] \sqcup σ|_{X_{d_i}'} ∈ {X_{d_i}', X_{d_i}', X_{d_i}', X_{d_i}', X_{d_i}'}

By applying WIND to (A3) wherever m_i < max(m_1, ..., m_n), we have
(A4) ∀ i ∈ {1, 2, ..., n}, dpool ⊢ D_i(X_{d_i}), σ|_{X_{d_i}'} →_{max(m_1, ..., m_n)+1} d_i'(x_{d_i}'), [σ'_{i}(z_{d_i}')/x_{d_i}'] \sqcup σ|_{X_{d_i}'} ∈ {X_{d_i}', X_{d_i}', X_{d_i}', X_{d_i}', X_{d_i}'}

By Sigma Extension (Lemma 6),
∀ i ∈ {1, 2, ..., n}, [σ'_{i}(z_{d_i}')/x_{d_i}'] ≥ σ|_{X_{d_i}'}

Using (A4) and the above, we apply Unrolling Subdividings (Lemma 9) to obtain:
(A5) dpool ⊢ d'(x_{d_i}), σ|_{X_{d_i}'} →_{max(m_1, ..., m_n)+1} d_i'(x_{d_i}'), σ|x ∈ ∪_{i=1}^n[σ'_{i}(z_{d_i}')/x_{d_i}'],
where d(x_{d_i}) = (rD_i, p(x), (D_i(X_{d_i}'), q_i(x_i)))::(D_i(X_{d_i}'), q_i(x_i))::nil
and d'(x_{d_i}') = (rD_i, p(x), (d_i'(x_{d_i}'), q_i(x_i)))::(d_i'(x_{d_i}'), q_i(x_i))::nil

Pick any σ' ≥ σ|x ∪ ∪_{i=1}^n[σ'_{i}(z_{d_i}')/x_{d_i}'] s.t. x_{d_i}' ⊆ dom(σ')

Using (N8) and (N9),
Pick B = \bigcup_{i=1}^n B_i.

By construction,
σ' ≥ σ

Using the above and (A),
(A6) ⊢ c_{pu}(\tilde{x}, x_1', ..., x_n')σ

By construction,
∀ i ∈ {1, 2, ..., n}, σ' ≥ [σ'_{i}(z_{d_i}')/x_{d_i}'] ≥ [σ'_{i}(z_{d_i}')/x_{d_i}']

By (N8), (N9), and the above,
∀ i ∈ {1, 2, ..., n}, prog, B_i ⊢ d_i'(x_{d_i}')σ':q_i(x_i)σ'
Since B_i ⊆ B,
(A7) ∀ i ∈ {1, 2, ..., n}, prog, B ⊢ d_i'(x_{d_i}')σ':q_i(x_i)σ'

By (A6) and (A7), we can apply rD to obtain:
(A8) prog, B ⊢ (rD, p(x), (d_i'(x_{d_i}'):q_i(x_i))::(d_i'(x_{d_i}'):q_i(x_i))::nil)σ':p(x)σ'

Case B: ∃i ∈ {1, 2, ..., n} s.t. the second condition in (N8) and (N9) holds.

By assumption,
(B1) Exists an index set Λ = {λ_1, ..., λ_n} with properties:
Λ ≠ ∅
1 ≤ λ_i ≤ n
the second condition in (N8) and (N9) holds

By (B1),
(B2) ∀j ∈ \{1, 2, ..., n\} \setminus \Lambda, the first condition in (N8) and (N9) holds

First, we show that ∃d'(z_{d'}), \sigma' s.t. dpool ⊢ d(x_{d'}), \sigma →_d d'(x_{d'}), \sigma'.

By (B1),
(B3) ∀\lambda_i ∈ \Lambda, dpool ⊢ D_{\lambda_i}(x_{Z_{\lambda_i}}), \sigma_i →_k d'_{\lambda_i}(z_{\lambda_i}), \sigma_i'

By (N6),
∀\lambda_i ∈ \Lambda, Z_{\lambda_i} ⊆ z_{\lambda_i}', and x_{\lambda_i}' are fresh variables for z_{\lambda_i}', X_{\lambda_i} ⊆ x_{\lambda_i}'.

By the above and (B3), using Substitution (Lemma 13), we have:
(B4) ∀\lambda_i ∈ \Lambda, dpool ⊢ D_{\lambda_i}(x_{Z_{\lambda_i}}), \sigma_i |_{X_{\lambda_i}} →_k d'_{\lambda_i}(x_{\lambda_i}), [\sigma'_{\lambda_i}(z_{\lambda_i}')/x_{\lambda_i}]

Since ∀i ∈ \{1, 2, ..., n\}, z_{d'} are fresh, ∀\lambda_i ∈ \Lambda, \sigma_i |_{x_{\lambda_i}}, ..., x_{\lambda_i+1}, x_{\lambda_i+1}, ..., x_{\lambda_i} is well-defined.

Using the above and Weakening (Lemma 11) to (B5),
(B5) ∀\lambda_i ∈ \Lambda, dpool ⊢ D_{\lambda_i}(x_{Z_{\lambda_i}}), \sigma_i |_{X_{\lambda_i}} →_k d'_{\lambda_i}(x_{\lambda_i}), [\sigma'_{\lambda_i}(z_{\lambda_i}')/x_{\lambda_i}]

By applying Weakto (B5) whenever m_{\lambda_i} < k, we have
(B6) ∀\lambda_i ∈ \Lambda, dpool ⊢ D_{\lambda_i}(Z_{\lambda_i}), \sigma_j →_k d'_{\lambda_i}(z_{\lambda_i}'), \sigma_j'

By (B2) and (N8), (N9),
(B7) ∀j ∈ \{1, 2, ..., n\} \setminus \Lambda, dpool ⊢ D_j(Z_{\lambda_j}), \sigma_j →_k d'_{\lambda_j}(z_{\lambda_j}'), \sigma_j'

By (N6),
∀j ∈ \{1, 2, ..., n\} \setminus \Lambda, Z_{\lambda_j} ⊆ z_{\lambda_j}', \bar{x}_{\lambda_j}' are fresh variables for z_{\lambda_j}', and \bar{x}_{\lambda_j}' ⊆ x_{\lambda_j}'.

By the above and (B7), using Substitution (Lemma 13), we have:
(B8) ∀j ∈ \{1, 2, ..., n\} \setminus \Lambda, dpool ⊢ D_j(Z_{\lambda_j}), \sigma_j |_{X_{\lambda_j}} →_k d'_{\lambda_j}(x_{\lambda_j}), [\sigma'_{\lambda_j}(z_{\lambda_j}')/x_{\lambda_j}]

Since ∀j ∈ \{1, 2, ..., n\}, z_{d'} are fresh, ∀j ∈ \{1, 2, ..., n\} \setminus \Lambda, \sigma_j |_{x_{\lambda_j}}, ..., x_{\lambda_j+1}, x_{\lambda_j+1}, ..., x_{\lambda_j} is well-defined.

Using the above and Weakening (Lemma 11) to (B8),
(B9) ∀j ∈ \{1, 2, ..., n\} \setminus \Lambda, dpool ⊢ D_j(Z_{\lambda_j}), \sigma_j |_{X_{\lambda_j}} →_k d'_{\lambda_j}(x_{\lambda_j}), [\sigma'_{\lambda_j}(x_{\lambda_j})/x_{\lambda_j}]

By Sigma Extension (Lemma 6),
∀i ∈ \{1, 2, ..., n\}, [\sigma'_{\lambda_i}(z_{\lambda_i}')]/x_{\lambda_i}' ≥ \sigma |_{x_{\lambda_i}}

By the above, (B6) and (B9), we apply Unrolling Subderivations (Lemma 9) to obtain:
(B10) dpool ⊢ d(x_{\lambda_i}), \sigma →_d d'(x_{\lambda_i}), \sigma |_{x_{\lambda_i}} [\sigma'_{\lambda_i}(z_{\lambda_i}')]/x_{\lambda_i}'].

where d'(x_{\lambda_i}) = (rID, p(x), (d_i(x_{\lambda_i})::q_i(x_i))::...:(d_n(x_{\lambda_i}'):q_n(x_n))::nil).

Next we show there ∃B s.t.
If ∃rec(s(z_{\lambda})), \exists d'(x_{\lambda})', exists a concrete derivation from B for s,
the ∃d''(x_{\lambda})'', σ'' s.t. for some σ'' ≥ σ', \exists prog, B ⊢ d''(x_{\lambda}'')\sigma'' : p(x)σ''

By (N8) and (N9),
∀\lambda_i ∈ \Lambda, \exists s(z_{\lambda_i}) ∈ d'_{\lambda_i}(z_{\lambda_i}')

By (N6), apply the substitution [x_{\lambda_i}'/z_{\lambda_i}'] throughout the above expression,
∀\lambda_i ∈ \Lambda, \exists s(x_{\lambda_i}) ∈ d'_{\lambda_i}(x_{\lambda_i}')

By the above,
(B11) \exists s(x_{\lambda_i}) ∈ d'_{\lambda_i}(x_{\lambda_i}')

∀j ∈ \{1, 2, ..., n\} \setminus \Lambda, pick any σ_i' ≥ σ_i s.t. z_{\lambda_i}' = dom(σ_i')

By the above and (N8), (N9),
(B12) ∀j ∈ \{1, 2, ..., n\} \setminus \Lambda, ∃B_i' s.t. prog, B_i' ⊢ d'(z_{\lambda_i}'), σ_i' : q_i(x_i)σ_i'

Using (N8), (N9) and (B12),
(B13) Pick B = \bigcup_{i=1}^{n} B_i.
either i ∈ \Lambda and B_i = B_i,
or i ∈ \{1, 2, ..., n\} \setminus \Lambda and B_i = B_i'

Assume the following:
(B14) \exists rec(s(z_{\lambda}')) ∈ d'(x_{\lambda}'),
\[ \exists d_s(x^s), k_s, \sigma_s \text{ s.t.} \]
\[ d_{pool} \vdash (\text{rec, } s(x^s)), \sigma_s|s^s \mapsto_k s_d d_s(x^s) \sigma_s \]
\[ \text{and } \exists \sigma_s \geq \sigma_s \text{ s.t. } \text{prog, } B \vdash d_s(x^s) \sigma_s \]

By construction,
\[ \forall \lambda_i \in \Lambda, (\text{rec, } s(x^s)) \in d'_\lambda_i (x_{\lambda_i}, ') \text{ implies } (\text{rec, } s(x^s)) \in d'_d (x^s') \]

By (N6), \( z_{\lambda_i} \) are fresh variables for \( x_{\lambda_i} \).

By the above and using (IB14),
\[ \forall \lambda_i \in \Lambda, \]
\[ \forall (\text{rec, } s(z_i)) \in d'_\lambda_i (z_{\lambda_i}, '), \]
\[ d_{pool} \vdash (\text{rec, } s(z_i)), [\sigma(x^s)/z_i] \mapsto_k s_d d_s(z_{\lambda_i}), [\sigma_s(x^s)/z_{\lambda_i}] \]
\[ \text{and } \text{prog, } B \vdash d_s(z_{\lambda_i})[\sigma_s(x^s)/z_{\lambda_i}], s(z_i)[\sigma_s(x^s)/z_{\lambda_i}] \]

By (IB15) and (N8), (N9),
\[ \forall \lambda_i \in \Lambda, \]
\[ \exists d'_\lambda_i (z_{\lambda_i} ''), L_{\lambda_i}, \sigma'_\lambda_i \text{ s.t.} \]
\[ d_{pool} \vdash D_{\lambda_i} (Z_{\lambda_i} ''), \sigma_{\lambda_i} \]
\[ \sim_0 D_{\lambda_i} (Z_{\lambda_i} ''), \sigma_{\lambda_i} \]
\[ \text{...} \]
\[ \sim_k d'_{\lambda_i} (z_{\lambda_i} ''), \sigma_{\lambda_i} \]
\[ \text{...} \]
\[ \sim_{k+L_{\lambda_i}} d''_{\lambda_i} (z_{\lambda_i} ''), \sigma''_{\lambda_i} \]
\[ \text{and } \exists d''_{\lambda_i} \geq \sigma''_{\lambda_i} \text{ s.t. } \text{prog, } B \vdash d''_{\lambda_i} (z_{\lambda_i} '') \sigma''_{\lambda_i} \]

By applying Substitution (Lemma 13) to (IB16), we obtain:
\[ \forall \lambda_i \in \Lambda, \]
\[ d_{pool} \vdash D_{\lambda_i} (X_{\lambda_i} ''), \sigma_{X,\lambda_i} \]
\[ \sim_0 D_{\lambda_i} (X_{\lambda_i} ''), \sigma_{X,\lambda_i} \]
\[ \text{...} \]
\[ \sim_k d'_{\lambda_i} (x_{\lambda_i} ''), [\sigma'_{\lambda_i} (z_{\lambda_i} ') / x_{\lambda_i} '] \]
\[ \text{...} \]
\[ \sim_{k+L_{\lambda_i}} d''_{\lambda_i} (x_{\lambda_i} ''), [\sigma''_{\lambda_i} (z_{\lambda_i} '') / x_{\lambda_i} '] \]

Since \( \forall i \in \{1, 2, \ldots, n\} \), \( z_{\lambda_i} '' \) are fresh,
\[ \forall \lambda_i \in \Lambda, \sigma_{|x_{\lambda_i}, \ldots, x_{\lambda_i+1}, x_{\lambda_i+1}, \ldots, x_{\lambda_i+n}} \cup [\sigma''_{\lambda_i} (z_{\lambda_i} '') / x_{\lambda_i} '] \text{ is well-defined.} \]

Therefore by Weakening (Lemma 11), we obtain:
\[ \forall \lambda_i \in \Lambda, \text{ applying Rule } \text{WK10} \text{ where } L_{\lambda_i} < \max_{\lambda_i \in \Lambda} L_{\lambda_i} \text{ to obtain:} \]
\[ \text{By (B7), } \forall \lambda_i \in \Lambda, \text{ applying the Substitution (Lemma 13) to (B19), we obtain:} \]
\[ \forall \lambda_i \in \Lambda, \]
\[ d_{pool} \vdash D_{\lambda_i} (X_{\lambda_i} ''), \sigma_{|x_{\lambda_i}, \ldots, x_{\lambda_i+n}} \]
\[ \sim_0 D_{\lambda_i} (X_{\lambda_i} ''), \sigma_{|x_{\lambda_i}, \ldots, x_{\lambda_i+n}} \]
\[ \text{...} \]
\[ \sim_k d'_{\lambda_i} (x_{\lambda_i} ''), [\sigma'_{\lambda_i} (z_{\lambda_i} ') / x_{\lambda_i} '] \]
\[ \text{...} \]
\[ \sim_{k+\max_{\lambda_i \in \Lambda} L_{\lambda_i}} d''_{\lambda_i} (x_{\lambda_i} ''), [\sigma''_{\lambda_i} (z_{\lambda_i} '') / x_{\lambda_i} '] \]

By (B7),
\[ \forall j \in \{1, 2, \ldots, n\} \setminus \Lambda, \]
\[ d_{pool} \vdash D_j (Z_d'), \sigma_j \mapsto_{m_j} d'_j (z_d '), \sigma'_j \]
\[ \text{and } \text{prog, } B_j \vdash d'_j(z_d ') \sigma'_j \]

By applying the Substitution (Lemma 13) to (BI20), we obtain:
\[ \forall \lambda_i \in \Lambda, \]
\[ d_{pool} \vdash D_j (X_d'), \sigma_{|x_d} \mapsto_{m_j} d'_j (z_d '), [\sigma'_j (z_d ') / x_d '] \]

Since \( \forall i \in \{1, 2, \ldots, n\} \), \( z_d '' \) are fresh,
\[ \forall j \in \{1, 2, \ldots, n\} \setminus \Lambda, \sigma_{|x_{j}, \ldots, x_{j+1}, \ldots, x_{n}} \cup [\sigma'' (z_d ') / x_d '] \text{ is well-defined.} \]

Therefore by applying Weakening (Lemma 11) to (B21), we obtain:
\[ \forall j \in \{1, 2, \ldots, n\} \setminus \Lambda, \]
By assumption, $\Delta_p = (\rho_p(x_{p_1}, x_{c_{p_1}}), d_{p_1}(x_{p_1}, x_{d_{p_1}}); \rho_p(x_{p_2}, x_{c_{p_2}}), d_{p_2}(x_{p_2}, x_{d_{p_2}}); \ldots; (\rho_p(x_{p_m}, x_{c_{p_m}}), d_{p_m}(x_{p_m}, x_{d_{p_m}}); \rho_p(x_{p_{m+1}}), d_{p_{m+1}})) \vdash \nu_i$. 

By GENDS,

(R2) $\forall \sigma \vdash (c_{rec_p}(\bar{x})[\bar{u}]/\bar{x}) \Leftrightarrow \forall_{j=1}^{m} \exists x_{c_{p_j}}. \rho_{p_j}(x_{p_j}, x_{c_{p_j}})[\bar{u}/x_{p_j}])\sigma \vdash (c_{rec_p}(\bar{x})[\bar{u}]/\bar{x})$

$\bar{u}$ are fresh variables corresponding to the arguments of predicate $p$.

By (R1),

(R4) $c_{rec_p}(\bar{x})[\bar{u}]/\bar{x} \vdash (c_{rec_p}(\bar{x})[\bar{u}]/\bar{x})$

By (R3) and (R4),

(R5) $\exists_{j=1}^{m} \exists x_{c_{p_j}}. \rho_{p_j}(x_{p_j}, x_{c_{p_j}})[\bar{u}/x_{p_j}] \vdash (c_{rec_p}(\bar{x})[\bar{u}]/\bar{x})$

Pick $\sigma = [\sigma(\bar{x})/\bar{u}]$. Then by (R2), we have:

$\vdash (c_{rec_p}(\bar{x})[\bar{u}]/\bar{x}) \Leftrightarrow \forall_{j=1}^{m} \exists x_{c_{p_j}}. \rho_{p_j}(x_{p_j}, x_{c_{p_j}})[\bar{u}/x_{p_j}]\sigma(\bar{x})/\bar{u}$

Which is logically equivalent to:

(R3) $\Leftrightarrow (c_{rec_p}(\bar{x})[\bar{u}]/\bar{x}) \Leftrightarrow \forall_{j=1}^{m} \exists x_{c_{p_j}}. \rho_{p_j}(x_{p_j}, x_{c_{p_j}})[\bar{u}/x_{p_j}]\sigma(\bar{x})/\bar{u}$

By (R1),

(R4) $c_{rec_p}(\bar{x})[\bar{u}]/\bar{x} \vdash (c_{rec_p}(\bar{x})[\bar{u}]/\bar{x})$

By (R3) and (R4),

(R5) $\exists_{j=1}^{m} \exists x_{c_{p_j}}. \rho_{p_j}(x_{p_j}, x_{c_{p_j}})[\bar{u}/x_{p_j}] \vdash (c_{rec_p}(\bar{x})[\bar{u}]/\bar{x})$

By (R5), $\exists j \in \{1, 2, \ldots, m\}$ s.t. $\vdash (\exists x_{c_{p_j}}. \rho_{p_j}(x_{p_j}, x_{c_{p_j}})[\bar{u}/x_{p_j}])\sigma(\bar{x})/\bar{u}$

By the above,

(R6) $\exists x_{c_{p_j}}$ where $\operatorname{dom}(\sigma_{c_{p_j}}) = x_{c_{p_j}}$, s.t.

$\vdash (\exists x_{c_{p_j}}. \rho_{p_j}(x_{p_j}, x_{c_{p_j}})[\bar{u}/x_{p_j}])\sigma(\bar{x})/\bar{u}$

Because $\sigma_{c_{p_j}}$ is fresh, $\sigma \cup \sigma_{c_{p_j}}$ is well-defined.
(R7) \( dp_j(x_{pq}, x_{d_p}) = (rID, p(x_{pq}), (D_1(X_{d_1})q_1(x_1))::\ldots::(D_n(X_{d_n})q_n(x_n))::nil) \)
where \( x_{dpj} = \{X_{d_1}, \ldots, X_{d_n}\} \)

By (R7), the rule that derives \( p \)

By (R1) and (R6),

Using (R1), (R3), (R10), and (R12), we apply \( R \)

By (R5),

we apply the Induction Hypothesis to obtain:

\[ c \vdash dpool, \text{ and using (R5) and (R9),} \]

\[ (\text{rec, } q_i) \]

either \( q_i \) is not on a cycle in \( G \),

thus \( \exists (c_i(z_i'), d_i(z_i')q_i(z_i')) \in \text{dpool}(q_i) \), and

\[ D_i(X_{d_i})[Z_{d_i}/X_{d_i}] = d_i(z_i') \]

or \( q_i \) is on a cycle in \( G \),

thus \( \exists (c_{\text{rec}}(z_i'), A_{d_i}) \in \text{dpool}(q_i) \) and

\[ D_i(X_{d_i})[Z_{d_i}/X_{d_i}] = (\text{rec}, q_i(z_i')) \]

By (R8) \( \forall i \in \{1, 2, \ldots, n\}, \)

either \( q_i \) is not on a cycle

\[ \text{and } z_{d_i} \subseteq z_i' \]

or \( q_i \) is on a cycle

\[ \text{and } z_i' \subseteq z_{d_i}' \]

By (R1) and (R6),

(\text{R12}) \( \forall i \in \{1, 2, \ldots, n\}, \)

\[ C_i(Z_{ci}) \sigma_i, \text{ where } \sigma_i = [\sigma_{\text{cpj}}(X_{ci})]/Z_{ci}, \]

Pick \( w_{dpj} \) to be fresh variables corresponding to \( x_{dpj} \), where \( x_{dpj} = \{Z_{d_1}, \ldots, Z_{dn}\} \).

By (R5),

\[ w_{dpj} = \{\bar{W}_{d_1}, \ldots, W_{d_n} \} \]

and \( w_{dpj} \) are also fresh variables corresponding to \( \{Z_{d_1}, \ldots, Z_{dn}\} \)

By (R1) and (R4),

(\text{R14}) \[ c_{\text{cpj}}(x_{pq}, x_{d_p}, x_1, \ldots, x_n) | \bar{x}/x_{pq} | [w_{dpj}/x_{dpj}] | [\sigma \cup [\sigma_{\text{cpj}}(x_1, \ldots, x_n)]/w_1, \ldots, w_n] \]

Since \( w_1, \ldots, w_n \) are fresh,

\( \sigma \cup [\sigma_{\text{cpj}}(x_1, x_2, \ldots, x_n)]/w_1, \ldots, w_n \)

is well-defined

Using (R1), (R3), (R10), and (R12), we apply \text{Rrec} to obtain:

\[ \text{dpool} \vdash (\text{rec, } p(\bar{x})), \sigma \vdash_{\text{d}} d_{\sigma}(x_{dpj}, x_{dpj}) \]

\[ = dpool(q_i) \]

\[ \text{where } \sigma = [\sigma_{\text{cpj}}(X_{ci})]/Z_{ci}, \]

and since \( W_{c_1}, \ldots, W_{c_n} \) are fresh,

\( \sigma \cup [\sigma_{\text{cpj}}(x_1, x_2, \ldots, x_n)]/w_1, \ldots, w_n \)

is well-defined.

We can rewrite the above as:

(\text{R15}) \[ \text{dpool} \vdash (\text{rec, } p(\bar{x})), \sigma \]

\[ \vdash_{\text{d}} (rID, p(\bar{x}), (D_1(W_{d_1})q_1(w_1))::\ldots::(D_n(W_{d_n})q_n(w_n))::nil), \sigma \cup [\sigma_{\text{cpj}}(X_{ci})]/W_{ci} \]

Since \( k < \ell \), and using (R5) and (R9),

we apply the Induction Hypothesis to obtain:

\( \forall i \in \{1, 2, \ldots, n\}, \)

(\text{R16}) If \( (c_i(z_i'), d_i(z_i')q_i(z_i')) \in \text{dpool}(q_i) \), then

\( \exists m_i \leq k, d'_i(z_i'), \sigma'_i \text{ s.t.} \)

\[ \text{dpool} \vdash d_i(z_i'), \sigma_i \rightarrow_{m_i} d'_i(z_i'), \sigma'_i \]

and \( d'_i \) does not contain any \( (\text{rec, } s(z_i')) \)

and \( \forall \sigma'_i \geq \sigma_i \text{ where } z_{d_i}' \subseteq \text{dom}(\sigma'_i) \)

\[ \text{implies } \exists B_i \text{ s.t. prog, } B_i \vdash d'_i(z_i') \sigma'_i q_i(z_i') \]

or \( \exists d'_i(z_i'), \sigma'_i \text{ s.t.} \)

\[ d_i(z_i') \sigma_i \rightarrow_{k} d'_i(z_i'), \sigma'_i \]

\[ \text{and } \exists B_i \text{ s.t. prog, } B_i \vdash d'_i(z_i') \sigma_i s_i(z_i') \]

\[ \text{and } \exists B_i \text{ s.t.} \]

\[ \forall (\text{rec, } s(z_i')) \in d'_i(z_i'), \exists d_{\sigma}(z_{d_\sigma}), k, \sigma \text{ s.t.} \]

\[ \text{dpool} \vdash (\text{rec, } s(z_i')), \sigma_i | z_i' \rightarrow_{k} d_{\sigma}(z_{d_\sigma}), \sigma_{\sigma} \]

and \( \exists \sigma_{\sigma} \geq \sigma_i \text{ s.t. prog, } B_i \vdash d_{\sigma}(z_{d_\sigma}) \sigma_{\sigma} s_i(z_i') \sigma_{\sigma} \)

implies \( \exists d'_i(z_i'), L_i, \sigma''_i \text{ s.t.} \)

\[ \text{dpool} \vdash d_i(z_i'), \sigma_i \]

\[ \ldots \]

\[ \ldots \]
\[ \sim_k d'_i(z_{di})', \sigma'_i \]
\[ \vdots \]
\[ \sim_k+L_i d''_i(z_{di}''), \sigma''_i \]

and \[ \exists \sigma''_i \geq \sigma'_i \text{ s.t. } \text{prog}, B_i \vdash d''_i(z_{di}'') \sigma''_i : q_i(\vec{x}), \sigma''_i \]

(R17) If \((c_{i \in \text{rec}(z_{ci}'), \Delta_{qi}}) \in \text{dpool}(q_i)\), then:

- Either \(\exists m_i \leq k, d'_i(z_{di}'), \sigma'_i \text{ s.t. } \)
- \(\text{dpool} \vdash (\text{rec, } q_i(z_{zi})), \sigma_i \rightarrow_m d'_i(z_{di}'), \sigma'_i, \)
- and \(d'_i \) does not contain any \(\text{rec, } s(z_{zi})\),
- and \(\forall \sigma'_i \geq \sigma'_i \text{ where } z_{di}' \subseteq \text{dom}(\sigma'_i)\),
- \(B_i \text{ s.t. } \text{prob, } B_i \vdash d'_i(z_{di}') \sigma'_i : q_i(\vec{x}), \sigma'_i, \)
- or \(\exists d'_i(z_{di}'), \sigma'_i \text{ s.t. } \)
- \(\text{dpool} \vdash (\text{rec, } q_i(z_{zi})), \sigma_i \rightarrow_k d'_i(z_{di}'), \sigma'_i, \)
- and \(\exists (\text{rec, } s(x_{zi}')) \text{ s.t. } (\text{rec, } s(x_{zi}')) \in d'_i(z_{di}') \)
- and \(\exists B_i \text{ s.t. } \)
- \(\forall (\text{rec, } s(z_{zi}')) \in d'_i(z_{di}'), \)
- \(\exists d_s(z_{di}'), k_s, \sigma_s \text{ s.t. } \)
- \(\text{dpool} \vdash (\text{rec, } s(z_{zi}')), \sigma'_i |_{z_{zi}'} \rightarrow_k d_s(z_{di}'), \sigma_s \)
- and \(\exists \sigma_s \geq \sigma_s \text{ s.t. } \text{prob, } B_i \vdash d_s(z_{di}') \sigma_s : s(z_{zi}) \sigma_s \)

implies

\[ \exists d''_i(z_{di}''), L_i, \sigma''_i \text{ s.t. } \]
\[ \text{dpool} \vdash (\text{rec, } q_i(z_{zi}')), \sigma_i \]
\[ \vdots \]
\[ \sim_k d'_i(z_{di}'), \sigma'_i \]
\[ \vdots \]
\[ \sim_k+L_i d''_i(z_{di}''), \sigma''_i \]

and \(\exists \sigma''_i \geq \sigma'_i \text{ s.t. } \text{prob, } B_i \vdash d''_i(z_{di}'') \sigma''_i : q_i(\vec{x}), \sigma''_i \)

We need to consider two cases:

(C) \(\forall i \in \{1, 2, \ldots, n\}, \) the first condition in (R16) and (R17) holds,

(D) \(\exists i \in \{1, 2, \ldots, n\} \text{ s.t. the second condition in (R16) and (R17) holds.} \)

(C) \(\forall i \in \{1, 2, \ldots, n\}, \) the first condition in (R16) and (R17) holds

By (R14) and (R15),

(C1) \(\forall i \in \{1, 2, \ldots, n\}, \) \(\text{dpool} \vdash D_i(Z_{di})', \sigma_i \rightarrow_{m_i} d'_i(z_{di}'), \sigma'_i, \)

either \(q_i \) is not on a cycle and \(D_i(Z_{di}) = d_i(z_{di}) \)

or \(q_i \) is on a cycle and \(D_i(Z_{di}) = (\text{rec, } q_i(z_{zi}')) \)

By (R8), (R10), (R14), (R15), and (C1),

\(\forall i \in \{1, 2, \ldots, n\}, Z_{di} \subseteq z_{di}', w_{di}' \) are fresh variables corresponding to \(z_{di}' \), and \(W_{di} \subseteq w_{di}' \).

By the above, we can apply the Substitution (Lemma 13) to obtain:

(C2) \(\forall i \in \{1, 2, \ldots, n\}, \)
\[ \text{dpool} \vdash D_i(W_{di}), [\sigma_i(Z_{ci})/W_{ci}] \rightarrow_{m_i} d'_i(w_{di}'), [\sigma'_i(z_{di}')/w_{di}'] \]

Since \(\forall i \in \{1, 2, \ldots, n\}, w_{di}' \) are fresh,

\(\forall i \in \{1, 2, \ldots, n\}, \sigma \cup \bigcup_{j \neq i} [\sigma_i(Z_{di})/W_{di}] \) is well-defined.

Using the above, we apply Weakening (Lemma 11) to (A2) to obtain:

(C3) \(\forall i \in \{1, 2, \ldots, n\}, \)
\[ \text{dpool} \vdash D_i(W_{di}), \sigma \cup \bigcup_{j \neq i} [\sigma_j(Z_{di})/W_{di}] \rightarrow_{m_i} d'_i(w_{di}'), [\sigma'_i(z_{di}')/w_{di}'] \cup (\sigma \cup \bigcup_{j \neq i} [\sigma_j(Z_{di})/W_{di}]) \]

By applying \textsf{Weak} to (C3) wherever \(m_i < \max(m_1, \ldots, m_n) \), we have

(C4) \(\forall i \in \{1, 2, \ldots, n\}, \)
\[ \text{dpool} \vdash D_i(W_{di}), \sigma \cup \bigcup_{j \neq i} [\sigma_j(Z_{di})/W_{di}] \rightarrow_{\max(m_1, \ldots, m_n)} d'_i(w_{di}'), [\sigma'_i(z_{di}')/w_{di}'] \cup (\sigma \cup \bigcup_{j \neq i} [\sigma_j(Z_{di})/W_{di}]) \]

By Sigma Extension (Lemma 6),

\(\forall i \in \{1, 2, \ldots, n\}, [\sigma_i(z_{di}')/w_{di}'] \geq [\sigma_i(Z_{di})/W_{di}] \)

Using (R13) and (C4), we apply Unrolling Subderivations from Recursive Predicate (Lemma 8) to obtain:

(C5) \(\text{dpool} \vdash (\text{rec, } p(\vec{x})), \sigma \)
\[ \sim_{\sigma_0} (\text{rec, } p(\vec{x})), \sigma \]
\[\sim_1 (rID, p(\vec{x})), (D_1(W_{d1}); q_1(w_1)); \ldots; (D_n(W_{dn}); q_n(w_n)); \text{nil}), \sigma \sqcup \bigcup_{i=1}^{n}[\sigma_i(Z_{di})/W_{di}]\]
\[\ldots \sim_{\max(m_1, \ldots, m_n)+1} (rID, p(\vec{x}), (d'_1(w_{d1}'); q_1(w_1')); \ldots; (d'_n(w_{d1}); q_n(w_n)); \text{nil}), \sigma \sqcup \bigcup_{i=1}^{n}[\sigma'_i(z_{di}')/w_{di}'].\]

Pick any \(\vec{\sigma}' \geq \sigma \sqcup \bigcup_{i=1}^{n}[\sigma'_i(z_{di}')/w_{di}']\) s.t. \(\text{dom}(\vec{\sigma}') \subseteq \{\vec{x}, w_{d1}', \ldots, w_{dn}'\}\).

Using (N14) and (N15),

\[\text{Pick } B = \bigcup_{i=1}^{n} B_i.\]

By construction,

\[\vec{\sigma}' \geq \sigma\]

Using the above and (R12),

\[(C6) \vdash c_{rID}(\vec{x}, w_{d1}, \ldots, w_{dn})\vec{\sigma}'\]

By construction,

\[\forall i \in \{1, 2, \ldots, n\}, \sigma' \geq [\sigma'_i(z_{di}')/w_{di}'] \geq [\sigma'_i(z_{di}')/w_{di}']\]

By (N8), (N9), and the above,

\[\forall i \in \{1, 2, \ldots, n\}, \text{prog}, B_i \vdash d'_i(w_{d1}')\vec{\sigma}' : q_i(w_i)\vec{\sigma}'\]

Since \(B_i \subseteq B\),

\[(C7) \forall i \in \{1, 2, \ldots, n\}, \text{prog}, B \vdash d'_i(w_{d1}')\vec{\sigma}' : q_i(w_i)\vec{\sigma}'\]

By (C6) and (C7), we can apply \(rID\) to obtain:

\[(C8) \text{prog}, B \vdash (rID, p(\vec{x})), (d'_1(w_{di}'); q_1(w_1')); \ldots; (d'_n(w_{di}); q_n(w_n)); \text{nil})\vec{\sigma}' : p(\vec{x})\vec{\sigma}'\]

Case D: \(\vec{z}_i \in \{1, 2, \ldots, n\}\) s.t. the second condition in (N14) and (N15) holds.

By assumption,

(D1) Exists an index set \(\Lambda = \{\lambda_1, \ldots, \lambda_n\}\) with properties:
\[\Lambda \neq \emptyset\]
\[1 \leq \lambda_i \leq n\]
the second condition in (N8) and (N9) holds

By (D1),

(D2) \(\forall j \in \{1, 2, \ldots, n\}\setminus \Lambda\), the first condition in (N14) and (N15) holds

First, we show that \(\exists d'(\vec{x}'_i), \vec{\sigma}' \) s.t. \(dpool \vdash (\text{rec}, p(\vec{x})), \vec{\sigma} \mapsto d'(\vec{x}'_i)), \vec{\sigma}'\).

By (D1),

(D3) \(\forall \lambda_i \in \Lambda, dpool \vdash D_{\lambda_i}(Z_{\lambda_i}), \sigma_{\lambda_i} \mapsto_k d'_i(z_{\lambda_i}'), \sigma_{\lambda_i}'\)

By (N6),

\(\forall \lambda_i \in \Lambda, Z_{\lambda_i} \subseteq z_{\lambda_i}', w_{\lambda_i} \subseteq w_{\lambda_i}'\) are fresh variables corresponding to \(z_{\lambda_i}', \) and \(W_{\lambda_i} \subseteq w_{\lambda_i}'.\)

By the above and (B3), using Substitution (Lemma 13), we have:

\[(D4) \forall \lambda_i \in \Lambda, dpool \vdash D_{\lambda_i}(W_{\lambda_i}), \sigma_{\lambda_i} \mapsto_k d'_i(x_{\lambda_i}'), [\sigma_{\lambda_i}'(z_{\lambda_i}')/x_{\lambda_i}']\]

Since \(\forall \lambda_i \in \Lambda, w_{\lambda_i}' \) are fresh,

\[\left\{\sigma \cup \bigcup_{j=1, j \neq \lambda_i}^{n}[\sigma_j(Z_{dj})/W_{dj}]\right\} \sqcup [\sigma_{\lambda_i}'(z_{\lambda_i}')/w_{\lambda_i}']\]

is well-defined.

Using the above and Weakening (Lemma 11) to (B5),

\[(D5) \forall \lambda_i \in \Lambda, \]
\[dpool \vdash D_{\lambda_i}(W_{\lambda_i}), \sigma \cup \bigcup_{j=1, j \neq \lambda_i}^{n}[\sigma_j(Z_{dj})/W_{dj}] \]
\[\mapsto_k m_{\lambda_i} \sigma_{\lambda_i}'(w_{\lambda_i}), [\sigma_{\lambda_i}'(z_{\lambda_i}')/w_{\lambda_i}']] \sqcup (\sigma \cup \bigcup_{j=1, j \neq \lambda_i}^{n}[\sigma_j(Z_{dj})/W_{dj}]\)\]

By applying \(W\text{Kind}\) to (B5) wherever \(m_{\lambda_i} < k\), we have

\[(D6) \forall \lambda_i \in \Lambda, \]
\[dpool \vdash D_{\lambda_i}(W_{\lambda_i}), \sigma \cup \bigcup_{j=1, j \neq \lambda_i}^{n}[\sigma_j(Z_{dj})/W_{dj}] \]
\[\mapsto_k d'_i(w_{\lambda_i}'), [\sigma_{\lambda_i}'(z_{\lambda_i}')/w_{\lambda_i}'] \sqcup (\sigma \cup \bigcup_{j=1, j \neq \lambda_i}^{n}[\sigma_j(Z_{dj})/W_{dj}]\)

By (D2) and (N8), (N9),

\[(D7) \forall j \in \{1, 2, \ldots, n\}\setminus \Lambda, dpool \vdash D_j(Z_{dj}), \sigma_j \mapsto_k d'_j(z_{dj}'), \sigma_j'\]

By (N6),

\(\forall j \in \{1, 2, \ldots, n\}\setminus \Lambda, Z_{dj} \subseteq z_{dj}', w_{dj}' \) are fresh variables corresponding to \(z_{dj}', \) and \(W_{dj} \subseteq w_{dj}'.\)

By the above and (B7), using Substitution (Lemma 13), we have:

\[(D8) \forall j \in \{1, 2, \ldots, n\}\setminus \Lambda, dpool \vdash D_j(W_{dj}), [\sigma_j(Z_{dj})/W_{dj}] \mapsto_k d'_j(w_{dj}'), [\sigma_j(z_{dj}')/w_{dj}']\]

By (N5),

\(\forall j \in \{1, 2, \ldots, n\}\setminus \Lambda, \sigma \cup \bigcup_{i=1, i \neq j}^{n}[\sigma_i(Z_{di})/W_{di}] \sqcup [\sigma_j(z_{dj}')/w_{dj}']\) is well-defined.

65
Using the above and Weakening (Lemma 11) to (D8),
(D9) \( \forall j \in \{1, 2, \ldots, n\} \setminus \Lambda, \)
\[
dpool \vdash D_i(W_{\bar{y}_j^0}), \sigma \sqcup \bigsqcup_{i=1}^n [\sigma_i(Z_{\bar{d}_i}^0)/W_{\bar{d}_i}]
\]
\[
\iff \lambda \; d_j'(w_{\bar{y}_j^0}), \lambda \; d_j'(\bar{z}_{\bar{d}_i}^0)/w_{\bar{y}_j^0} \sqcup (\sigma \sqcup \bigsqcup_{i=1, i \neq j}^{n} [\sigma_i(Z_{\bar{d}_i}^0)/W_{\bar{d}_i}])
\]

Using (D6) and (D9), we apply Unrolling Subderivations (Lemma 9) to obtain:

(D10) \[
dpool \vdash (rID, p(x), (D_i(W_{\bar{d}_i}^0); \sigma_i(Z_{\bar{d}_i}^0)/W_{\bar{d}_i}))) \colon \vdots \vdash (D_n(W_{\bar{d}_n}^0); q_n(w_{\bar{w}_n}^0)); \sigma_n(Z_{\bar{d}_n}^0)/W_{\bar{d}_n}^0)
\]
\[
\iff \lambda \; d_j'(w_{\bar{y}_j^0}), \lambda \; d_j'(\bar{z}_{\bar{d}_i}^0)/w_{\bar{y}_j^0} \sqcup (\sigma \sqcup \bigsqcup_{i=1}^{n} [\sigma_i(Z_{\bar{d}_i}^0)/W_{\bar{d}_i}])
\]

Next we show \( \exists B \) s.t.

If \( \forall (\text{rec}, s(z_i^0)) \in d'(x_i^0), \) exists a concrete derivation for \( B \) from \( s, \)

the \( \exists d'(x_i^0), \lambda') \) s.t. for some \( \lambda' \geq \lambda', \) prog, \( B \vdash d'(x_i^0) \sigma':\text{prog} \)

By (N8) and (N9),
\( \forall \lambda_i \in \Lambda, \exists s(z_i^0) \in d'_i(z_{\bar{d}_i}^0) \)

By (N6), apply the substitution \[ w_{\bar{d}_i}^0/z_{\bar{d}_i}^0 \] throughout the above expression,
\( \forall \lambda_i \in \Lambda, \exists s(w_{\bar{w}_i}^0) \in d'_i(w_{\bar{w}_i}^0) \)

By the above,
(D11) \( \exists s(w_{\bar{w}_i}^0) \in (rID, p(x), (d'_i(w_{\bar{w}_i}^0); q_1(w_{\bar{w}_i}^0)); \vdots \vdash (d'_n(w_{\bar{w}_n}^0); q_n(w_{\bar{w}_n}^0)); \text{nil}) \)

\( \forall j \in \{1, 2, \ldots, n\} \setminus \Lambda, \) pick any \( \lambda_j' \geq \lambda_j \) s.t. \( \bar{z}_{\bar{d}_i}^0 = \text{dom}(\lambda_j') \)

By the above and (R14), (R15),
(D12) \( \forall j \in \{1, 2, \ldots, n\} \setminus \Lambda, \exists B_j \) s.t. prog, \( B_j \vdash d'_j(z_{\bar{d}_i}^0) \sigma_j'; q_j(x_j^0) \sigma_j' \)

Using (N8), (N9) and (D12),

(D13) Pick \( B = \bigcup_{i=1}^{n} B_i, \)

either \( i \in \Lambda \) and \( B_i = B, \)

or \( i \in \{1, 2, \ldots, n\} \setminus \Lambda \) and \( B_i = B_i' \)

Assume the following:

(D14) \( \forall (\text{rec}, s(w_{\bar{w}_i}^0)) \in (rID, p(x), (d'_i(w_{\bar{w}_i}^0); q_1(w_{\bar{w}_i}^0)); \vdots \vdash (d'_n(w_{\bar{w}_n}^0); q_n(w_{\bar{w}_n}^0)); \text{nil}), \)

\( \exists d_i(w_{\bar{w}_i}^0), k_i, \sigma_s \) s.t.
\[
dpool \vdash (\text{rec}, s(w_{\bar{w}_i}^0)), (\sigma \sqcup \bigsqcup_{i=1}^{n} [\sigma_i'(z_{\bar{d}_i}^0)/w_{\bar{w}_i}^0]));w_i \iff k_i \; d_s(w_{\bar{w}_i}^0), \sigma_s \)

and \( \exists \sigma_s \geq \sigma_s \) s.t. prog, \( B \vdash d_s(w_{\bar{w}_i}^0) \sigma_s \)

By construction,
\( \forall \lambda_i \in \Lambda, \)
\( \text{rec}, s(w_{\bar{w}_i}^0) \in d'_i(w_{\bar{w}_i}^0) \)

implies
\( \text{rec}, s(w_{\bar{w}_i}^0)) \in (rID, p(x), (d'_i(w_{\bar{w}_i}^0); q_1(w_{\bar{w}_i}^0)); \vdots \vdash (d'_n(w_{\bar{w}_n}^0); q_n(w_{\bar{w}_n}^0)); \text{nil}) \)

By (N6), \( z_{\bar{d}_i}^0 \) corresponds to \( w_{\bar{w}_i}^0 \)

By the above and using (D14),
(D15) \( \forall \lambda_i \in \Lambda, \)
\( \forall (\text{rec}, s(z_i^0)) \in d'_i(z_{\bar{d}_i}^0), \)
\[
dpool \vdash (\text{rec}, s(z_i^0)), (\sigma'_i|z_i \iff \lambda_i \; d_s(z_{\bar{d}_i}^0), [\sigma_s(w_{\bar{w}_i}^0)/z_{\bar{d}_i}^0] \text{ s.t.} \text{prog, } B \vdash d_s(z_{\bar{d}_i}^0)|\sigma_s(w_{\bar{w}_i}^0)/z_{\bar{d}_i}^0] \)

By (D15) and (R16), (R17),
(D16) \( \forall \lambda_i \in \Lambda, \)
\( \exists d'_i(z_{\bar{d}_i}^0), L_{\lambda_i}, \sigma'_i \) s.t.
\[
dpool \vdash D_{\lambda_i}(Z_{\bar{d}_i}^0), \sigma'_{\lambda_i}
\]
\[
\iff 0 \; D_{\lambda_i}(Z_{\bar{d}_i}^0), \sigma_{\lambda_i}
\]
\[
\ldots
\]
\[
\iff k \; d'_{\lambda_i}'(z_{\bar{d}_i}^0), \sigma'_{\lambda_i}
\]
\[
\ldots
\]
\[
\iff k + L_{\lambda_i} \; d'_{\lambda_i}'(z_{\bar{d}_i}^0), \sigma'_{\lambda_i}
\]

By applying Substitution (Lemma 13) to (D16), we obtain:
(D17) \( \forall \lambda_i \in \Lambda, \)
\[
dpool \vdash D_{\lambda_i}(W_{\bar{d}_i}^0), [\sigma_{\lambda_i}(Z_{\bar{d}_i}^0)/W_{\bar{d}_i}]
\]
\[
\iff 0 \; D_{\lambda_i}(W_{\bar{d}_i}^0), [\sigma_{\lambda_i}(Z_{\bar{d}_i}^0)/W_{\bar{d}_i}]
\]
\[
\ldots
\]
\[
\iff k \; d'_{\lambda_i}(w_{\bar{w}_i}^0), [\sigma'_{\lambda_i}(z_{\bar{d}_i}^0)/w_{\bar{w}_i}^0]
\]
\[ \forall i \in \Lambda, D_{\lambda_i}(\vec{w}_{\sim \lambda_i}(\vec{w}_{\sim \lambda_i}')) \subseteq [\sigma'_{\lambda_i}(z_{\sim \lambda_i}''')/w_{\sim \lambda_i}'''] \]

Since \( w_{\sim \lambda_i}'''' \) is fresh.

Therefore by Weakening (Lemma 11), we obtain:

(D18) \( \forall \lambda \in \Lambda, \)

\[
dpool \vdash D_{\lambda_i}(W_{\sim \lambda_i}), \sigma \cup \prod_{j=1}^{n}[\sigma_j(Z_{\sim \lambda_i}/W_{\sim \lambda_i})] \}
\]

Since \( W_{\sim \lambda_i} \) and \( W_{\sim \lambda_i}'''' \) are fresh.

(D19) \( \forall j \in \{1, 2, \ldots, n\} \setminus \Lambda, \) applying Rule W Kind wherever \( L_{\lambda_i} < \max_{\lambda_i \in \Lambda} L_{\lambda_i} \) to obtain:

\[
dpool \vdash D_j(W_{\sim \lambda_i}), \sigma \cup \prod_{i=1}^{n}[\sigma_{i}(Z_{\sim \lambda_i}/W_{\sim \lambda_i})] \}
\]

By (D7)

(D20) \( \forall j \in \{1, 2, \ldots, n\} \setminus \Lambda, \)

\[
dpool \vdash D_j(Z_{\sim \lambda_i}), \sigma_j \rightarrow a_{m_j} d_j'(w_{\sim \lambda_i}'), \sigma_j'
\]

By applying the Substitution (Lemma 13) to (D20), we obtain:

(D21) \( \forall j \in \{1, 2, \ldots, n\} \setminus \Lambda, dpool \vdash D_j(W_{\sim \lambda_i}), \sigma_j(Z_{\sim \lambda_i}/W_{\sim \lambda_i}) \rightarrow a_{m_j} d_j'(w_{\sim \lambda_i}'), [\sigma_j'(z_{\sim \lambda_i}')/w_{\sim \lambda_i}''] \]

Since \( W_{\sim \lambda_i} \) and \( W_{\sim \lambda_i}'''' \) are fresh.

(D22) \( \forall j \in \{1, 2, \ldots, n\} \setminus \Lambda, \)

\[
dpool \vdash D_j(W_{\sim \lambda_i}), \sigma \cup \prod_{i=1}^{n}[\sigma_{i}(Z_{\sim \lambda_i}/W_{\sim \lambda_i})] \rightarrow a_{m_j} d_j'(w_{\sim \lambda_i}'), (\sigma \cup \prod_{i=1, i \neq j}^{n}[\sigma_{i}(Z_{\sim \lambda_i}/W_{\sim \lambda_i})]) \cup [\sigma_j'(z_{\sim \lambda_i}')/w_{\sim \lambda_i}''']
\]

By applying Rule W Kind to all the \( j \in \{1, 2, \ldots, n\} \setminus \Lambda \) in (D22),

(D23) \( \forall j \in \{1, 2, \ldots, n\} \setminus \Lambda, \)

\[
dpool \vdash D_j(W_{\sim \lambda_i}), \sigma \cup \prod_{i=1}^{n}[\sigma_{i}(Z_{\sim \lambda_i}/W_{\sim \lambda_i})] \}
\]

Using (D19) and (D22), apply Unrolling Subderivations from Recursive Predicate (Lemma 8) to obtain:

(D24) \( \)dpool \vdash (rec, p(\vec{\bar{x}})), \sigma

\[
\rightarrow_0 (rec, p(\vec{\bar{x}})), \sigma
\]

\[
\rightarrow_1 (rD_1, p(\vec{\bar{x}}), (D_1(W_{\sim \lambda_i}), q_1(w_{\bar{1}})) \vdots \vdots (D_n(W_{\sim \lambda_i}), q_n(w_{\bar{n}}))):nil, \]

\[
\sigma \cup \prod_{i=1}^{n} [\sigma_i(Z_{\sim \lambda_i}/W_{\bar{n}})]
\]

\[
\rightarrow_{k+1} (rD_1, p(\vec{\bar{x}}), D_1'(W_{\sim \lambda_i}'), q_1(w_{\bar{1}})) \vdots \vdots (D_n'(W_{\sim \lambda_i}'), q_n(w_{\bar{n}}))):nil, \]

\[
\sigma \cup \prod_{i=1}^{n} [\sigma_i'(z_{\sim \lambda_i}'')/w_{\bar{n}}']
\]

\[
\rightarrow_{\max_{\lambda_i \in \Lambda} L_{\lambda_i} + 1} (rD_1, p(\vec{\bar{x}}), D_1''(W_{\sim \lambda_i}'''), q_1(w_{\bar{1}})) \vdots \vdots (D_n''(W_{\sim \lambda_i}'''), q_n(w_{\bar{n}}))):nil, \]

\[
\sigma \cup \prod_{i=1, i \neq j}^{n} [\sigma_i'(z_{\sim \lambda_i}'')/w_{\bar{n}}''']
\]

where \( \forall \lambda_i \in \Lambda, D_{\lambda_i}'(W_{\sim \lambda_i}''') = [\sigma_{\lambda_i}'(z_{\sim \lambda_i}'')/w_{\bar{n}}'''] \)

and \( \forall j \in \{1, 2, \ldots, n\} \setminus \Lambda, D_j'(W_{\sim \lambda_i}') = d_j'(w_{\bar{n}}') \)

(D25) Pick \( \sigma = \sigma \cup \prod_{i=1, i \neq j}^{n} [\sigma_i'(z_{\sim \lambda_i}'')/w_{\bar{n}}''']; \)

Since \( \vec{\bar{x}}, \forall \lambda_i \in \Lambda, x_{\sim \lambda_i} \) are fresh, \( \sigma \) is well-defined.

By the above, (D24), (D25), and Sigma Extension (Lemma 6),

\[
\sigma \geq \sigma \cup [\prod_{i=1, i \neq j}^{n} [\sigma_i'(z_{\sim \lambda_i}'')/w_{\bar{n}}''']; \cup [\prod_{i=1}^{n} [\sigma_i'(z_{\sim \lambda_i}'')/w_{\bar{n}}''']
\]
By (N14) and (N15), and since $B_\lambda \subseteq B$,

$$\forall \lambda \in \Lambda, \quad \text{prog}, B \models d''(w_\lambda(z_\lambda), w_\lambda) = \varphi_\lambda(w_\lambda) \land \varphi_\lambda(w_\lambda)$$

and $[\varphi''(z_\lambda, w_\lambda)] \subseteq \sigma$

By (D12), and since $B_\ell \subseteq B$,

$$\forall j \in \{1, 2, \ldots, n\} \setminus \Lambda, \quad \text{prog}, B \models d''(w_0, w_0) = q_j(w_0) \land \varphi''(w_0, w_0)$$

and $[\varphi''(z_\lambda, w_\lambda)] \subseteq \sigma$

By (A) and (D4),

$$c(p(x, w_1, \ldots, w_n))$$

By (D24), (D25), (D26), (D27), (D28), we can apply $rID$ to obtain:

$$\text{prog}, B \models (rID, p(x), D''(W_{d_0}) \models q_0(w_0), \ldots, D''(W_{d_n}) : q_n(w_n))$$

$\square$

### B.3 Definitions, Lemmas, and Proof of Soundness Of Property Query

#### B.3.1 Definitions used in Soundness Of Property Query

We need an additional definition to prove the lemmas for the recursive case. Predicate appeared in derivation list (Definition 5) tells us that given a list of derivations for predicates $p_1, \ldots, p_n$, $q(W_q)$ appeared in one of the derivations for some $p_i$.

The following definition differs from Predicate appeared in the past (Definition 2), as it is possible that $q(W_q)$ is equal to some $p_i(W_i)$, thus $q(W_q)$ was not necessarily generated in a past derivation.

**Definition 5** (Predicate appeared in derivation list).

$$(D_1(W_{d_1}) ; p_1(W_1)) ; (D_2(W_{d_2}) ; p_2(W_2)) ; \ldots ; (D_n(W_{d_n}) ; p_n(W_n)) ; \models q(W_q)$$

iff $\exists i \in \{1, 2, \ldots, n\}$ s.t. $q(W_q) \in D_i(W_{d_i})$.

#### B.3.2 Lemmas used in the proof of Soundness Of Property Query

**Lemma 6** (Past Skeleton Derivations Onestep). If $dpool \vdash d_s(x_{d_s}, \sigma) \rightsquigarrow \ell d_{\ell+1}(x_{d_{\ell+1}}, \sigma_{\ell+1})$, then $\forall q(x_q)$, $d_s(x_{d_s}, \sigma) \models q(x_q)$ implies $d_{\ell+1}(x_{d_{\ell+1}}, \sigma_{\ell+1}) \models q(x_q)$

**Proof:**

Proof by induction on the derivation of $dpool \vdash d_s(x_{d_s}, \sigma) \rightsquigarrow \ell d_{\ell+1}(x_{d_{\ell+1}}, \sigma_{\ell+1})$.

**Rule BASE:**

$$\text{Rule BASE:}$$

\[
\text{BASE} \quad \frac{}{\text{dpool} \vdash d_0(x_{d_0}, \sigma_0) \rightsquigarrow \ell d_0(x_{d_0}, \sigma_0)}
\]

By the rule,

(B1) $d_s(x_{d_s}) = d_0(x_{d_0})$

and $d_{\ell+1}(x_{d_{\ell+1}}) = d_0(x_{d_0})$

(B2) Pick any $q(x_q)$.

Assume $d_0(x_{d_0}) : \models q(x_q)$.

By (B1) and (B2),

(B3) $d_0(x_{d_0}) : \models q(x_q)$.

**Rule WKIND:**

\[
\text{Rule WKIND:} \quad \frac{\text{dpool} \vdash d_s(x_{d_s}, \sigma) \rightsquigarrow k d_s(x_{d_s}, \sigma), \quad k < \ell \quad \text{dpool} \vdash d_{\ell}(x_{d_{\ell}}, \sigma) \rightsquigarrow \ell d_{\ell}(x_{d_{\ell}}, \sigma)}{\text{dpool} \vdash d_{\ell}(x_{d_{\ell}}, \sigma) \rightsquigarrow \ell d_{\ell}(x_{d_{\ell}}, \sigma)}
\]

By the rule,

(W1) $d_s(x_{d_s}) = d_{\ell+1}(x_{d_{\ell+1}})$

(W2) Pick any $q(x_q)$.

Assume $d_{\ell+1}(x_{d_{\ell+1}}) : \models q(x_q)$

By (W1) and (W2),

(W3) $d_s(x_{d_s}) : \models q(x_q)$

**Rule RNREC:**
Proof.

By assumption, Lemma 7 (Strengthening Past Derivations)

There are two cases to consider:

(a) either \( p(\bar{x}) = q(\bar{x}_q) \), or
(b) \( p(\bar{x}) \neq q(\bar{x}_q) \).

Case (a): \( p(\bar{x}) = q(\bar{x}_q) \).

By assumption,

\( (rID, p(\bar{x}), d_1', \ldots : d_m'::nil) : nil \models q(\bar{x}_q) \)

Case (b): \( p(\bar{x}) \neq q(\bar{x}_q) \).

By assumption,

\( (rID, p(\bar{x}), d_1', \ldots : d_m'::nil) : nil \models q(\bar{x}_q) \)

By the rule

\( d(x_{\bar{x},\ell}) = (rID, p(\bar{x}), d_1', \ldots : d_m'::nil) \)

and \( d_{\ell+1}(x_{\bar{x},\ell+1}) = (rID, p(\bar{x}), d_1', \ldots : d_m'::nil) \)

(N2) Pick any \( q(\bar{x}_q) \).

Assume \( (rID, p(\bar{x}), d_1', \ldots : d_m'::nil) : nil \vdash q(\bar{x}_q) \).

Rule RREC:

\[
dpool(p) = (c_p, \Delta_p) \\
\text{RREC} \\
dpool \vdash (\text{rec}, p(\bar{x})), \sigma_\ell \vdash (c(\bar{z}), \text{di}(\bar{z}); p(\bar{z})) \in \Delta_p \\
\text{fresh}(\bar{z}) = \bar{z}' \\
\vdash c(\bar{z})[\bar{z}'/(\bar{z}\backslash \bar{z})][\bar{x}/\bar{z}][\sigma_\ell \cup \sigma'] \\
dpool \vdash (\text{rec}, p(\bar{x})), \sigma_\ell \vdash 1 \quad \text{d}(\bar{z})[\bar{z}'/(\bar{z}\backslash \bar{z})][\bar{x}/\bar{z}], \sigma_\ell \cup \sigma' \\

(R1) Pick any \( q(\bar{x}_q) \).

Assume \( (\text{rec}, p(\bar{x})): \text{nil} \vdash q(\bar{x}_q) \).

By (R1),

\( p(\bar{x}) = q(\bar{x}_q) \)

By GENDPOOL and GENDRULE, \( d(\bar{z}_q) \) has form

\( (rID, p(\bar{z}), \text{dl}(\bar{z}_q\backslash \bar{z})) \)

By (R3) and Rule RREC,

\( (rID, p(\bar{z}), \text{dl}(\bar{z}_q\backslash \bar{z}))[\bar{z}'/(\bar{z}\backslash \bar{z})][\bar{x}/\bar{z}] = (rID, p(\bar{x}), \text{dl}(\bar{z}_q')) \)

By (R2) and (R4),

\( d(\bar{z}_q)[\bar{z}'/(\bar{z}_q\backslash \bar{z})][\bar{x}/\bar{z}] \vdash q(\bar{x}_q) \)

Lemma 7 (Strengthening Past Derivations). If \( (d(\bar{x}_q); p(x_p')) : \text{nil} \vdash q(\bar{x}_q) \) and \( p(x_p') \neq q(\bar{x}_q) \), then \( (d(\bar{x}_q); p(x_p')) : \text{nil} \vdash q(\bar{x}_q) \).

Proof.

By assumption,

(1) \( (d(\bar{x}_q); p(x_p')) : \text{nil} \vdash q(\bar{x}_q) \) and
(2) \( p(x_p') \neq q(\bar{x}_q) \)

Base Case: \( d(\bar{x}_q) = (BT, p(x_p)) \)
By (1),
(1) \((\text{BT}, p(x_p^0)): \text{nil} \vdash q(x_q^0)\)

By (1),
(2) \(q(x_q^0) \in (\text{BT}, p(x_p^0))\)

By (2),
(3) \(p(x_p^0) = q(x_q^0)\)

(3) contradicts (2)

---

### Inductive Case:

**Case A:** \(p\) is not on a cycle in \(G\)

By the semantics of \text{LookUp},

(a1) \(d(x_q^0) = (\text{rID}, p(x_p^0), (d_q(x_dq_1):q_1(x_1^1)):\ldots:(d_{qm}(x_dqm):q_m(x_m^m))::\text{nil})\)

where \(\{x_p^0, x_{d1}, \ldots, x_{dqm}\} = x_d\)

By (1) and (a1),
(a2) \((\text{rID}, p(x_p^0), (d_q(x_dq_1):q_1(x_1^1)):\ldots:(d_{qm}(x_dqm):q_m(x_m^m))::\text{nil})::p(x_p^0))::\text{nil} \vdash q(x_q^0)\)

By comparing (a2) and (2),
(a3) \(q(x_q^0) \in (d_q(x_dq_1):q_1(x_1^1)):\ldots:(d_{qm}(x_dqm):q_m(x_m^m))::\text{nil}\)

By (a3),
(a6) \(\exists i \in [1, m] \text{ s.t. } q(x_q^0) \in d_q(x_dq_i):q_i(x_i^i)\)

By (a6),
(a7) \(d_q(x_dq_i):q_i(x_i^i))::\text{nil} \vdash q(x_q^0)\)

In (a7),

Subcase 1: \(g(x_q^0) = g(x_i^i)\)

Subcase 2: \(g(x_q^0) \neq g(x_i^i)\)

Subcase 1: \(q(x_q^0) = g(x_i^i)\)

By (a7),
\(d_q(x_dq_i):q_i(x_i^i) \in (d_q(x_dq_1):q_1(x_1^1)):\ldots:(d_{qi}(x_dqi):q_i(x_i^i)):\ldots:(d_{qm}(x_dqm):q_m(x_m^m))::\text{nil}\)

By (2) and the above,
\((\text{rID}, p(x_p^0), (d_q(x_dq_1):q_1(x_1^1)):\ldots:(d_{qm}(x_dqm):q_m(x_m^m))::\text{nil})::p(x_p^0))::\text{nil} \vdash q(x_q^0)\)

Subcase 2: \(g(x_q^0) \neq g(x_i^i)\)

By (a7) and I.H.,
(a2.1) \(d_q(x_dq_i):q_i(x_i^i) \vdash q(x_q^0)\)

By the definition of “\(\vdash\),” and (a2.1),
(a2.2) \(\exists \sigma_y(x_dq_i) \text{ s.t. } d_q(x_dq_i):q(x_q^0) \in d_q(x_dq_i)\)

By (a2.2),
(a2.3) \(\exists \sigma_y(x_dq_i) \text{ s.t. } d_q(x_dq_i):q(x_q^0) \in (d_q(x_dq_1):q_1(x_1^1)):\ldots:(d_{qi}(x_dqi):q_i(x_i^i)):\ldots:(d_{qm}(x_dqm):q_m(x_m^m))::\text{nil}\)

By (2) and (a2.3),
\((\text{rID}, p(x_p^0), (d_q(x_dq_1):q_1(x_1^1)):\ldots:(d_{qm}(x_dqm):q_m(x_m^m))::\text{nil})::p(x_p^0))::\text{nil} \vdash q(x_q^0)\)

**Case B:** \(p\) is on a cycle in \(G\):

By the semantics of \text{LookUp},

(b1) \(d(x_q^0) = (\text{rec}, p(x_p^0))\)

By (1) and (b1),
(b2) \((\text{rec}, p(x_p^0)): \text{nil} \vdash q(x_q^0)\)

By (b2),
(b3) \(q(x_q^0) \in (\text{rec}, p(x_p^0))\)

By (b3),
(b4) \(q(x_q^0) = p(x_q^0)\)

(b4) contradicts (2)

---

**Lemma 8 (Past Skeleton Derivations),**

\(\forall k \in \{0, 1, \ldots, \ell - 1\},\)

\[d_{pool} \vdash d_q(x_{dq_0}), \sigma_0\]

\[\vdash_0 d_q(x_{dq_1}), \sigma_1\]

\[\vdots\]

\[\vdash_{k-1} d_k(x_{d_{k+1}}), \sigma_k\]
Proof.

Proof by induction on the structure of $d_k(x_{d,k})$.

Pick any $k \in \{0, 1, \ldots, \ell - 1\}$.

Assume

(1) $dpool \vdash d_0(x_{d,0}^0), \sigma_0$

(2) $d_{\ell-1}(x_{d,\ell-1}^1), \sigma_\ell$

(3) $d_k(x_{d,k}^0), \sigma_k$

(4) $d_{\ell-2}(x_{d,\ell-2}^1), \sigma_{\ell - 1}$

(5) $d_{\ell - 1}(x_{d,\ell - 1}^1), \sigma_{\ell - 1}$

(6) $d_k(x_{d,k}^0), \sigma_k$

(7) $d_k(x_{d,k}^0), \sigma_k$

(8) $d_k(x_{d,k}^0), \sigma_k$

(9) $d_k(x_{d,k}^0), \sigma_k$

Base Case: $d_k(x_{d,k}^0) = (BT, p(\vec{x}))$

Inductive Case

Subcase 1: $p$ is not on a cycle in $G$ and $d_k(x_{d,k}^0) = (rID, p(\vec{x}), (d_1(x_{q_1}^0), q_1(x_{q_1}^0)); \ldots; (d_m(x_{q_m}^0), q_m(x_{q_m}^0)); \nil)$

By (2),

(11.1) $\exists d_k(x_{q_k}^0)$ such that $d_k(x_{q_k}^0), q_k(x_{q_k}^0) \in (d_1(x_{q_1}^0), q_1(x_{q_1}^0)); \ldots; (d_m(x_{q_m}^0), q_m(x_{q_m}^0)); \nil$

By (11.1),

(11.2) $\exists j \in \{1, 2, \ldots, m\}$ such that $q(x_j) \in d_j(x_j)$

By (11.2), thus

(11.3) $(rID, p(\vec{x}), (d_1(x_{q_1}^0), q_1(x_{q_1}^0)); \ldots; (d_m(x_{q_m}^0), q_m(x_{q_m}^0)); \nil); p(\vec{x}) \vdash p(x_j)$

By (2),

$p(\vec{x}) = p_k(\vec{x})$

By the above and (1),

$p_k(\vec{x}) = p_{k+1}(x_{k+1})$

$p_{k+1}(x_{k+1}) = p_{k+2}(x_{k+2})$

$p_{\ell - 1}(x_{\ell - 1}) = p(\vec{x})$

Thus, we conclude

(11.4) $p_k(\vec{x}) = p(\vec{x}) = p(x)$

By (1), (11.3), (11.4), and repeated application of Past Skeleton Derivations OneStep (Lemma 6),

$d_k(x_{d,k}^0); p(\vec{x}) \vdash p(x_{d,k})$ implies $(d_i(x_{d,i})^0, p(\vec{x})); \nil \vdash p(x_{d,k})$

By (11.3) and the above,

(11.5) $(d_i(x_{d,i})^0, p(\vec{x})); \nil \vdash p(x_{d,k})$

By (4) and (11.4),
By (11.5), (11.6), and Strengthening Past Derivations (Lemma 7),
(11.7) \( (d_i(x_i^k);p(x_i)):\text{nil} \models_\nu q(x_i^k) \)

Subcase 2: \( p \) is on a cycle in \( G \) and \( d_k(x_{i,h}^k) = (\text{rec},p(x_i)) \)

By (4) and the definition of \( \models_\nu \),
(12.1) \( (\text{rec},p(x_i)):p(x_i)):\text{nil} \not\models_\nu q(x_i^k) \)
(12.1) contradicts (3)

\[ \square \]

### B.3.3 Proof of Soundness Of Property Query

**Theorem 6** (Soundness Of Property Query For Recursive Case).

\( \varphi = \forall \vec{x}_1.p_1(x_1) \land \ldots \land \forall \vec{x}_n.p_n(x_n) \land p_1(x_1, \ldots, x_n) \supset \exists y_1.q_1(y_1) \land \ldots \land \exists y_m.q_m(y_m) \land q(x_1, \ldots, y_1, \ldots, y_m) \)

DG\text{GRAPH}(prog) = G \text{ and GENDPOOL}(G,A) = dpool and
CKPROP(dpool, \varphi) = valid implies \( \forall B, \text{prog}, B \models \varphi \).

**Proof:**

By assumption,
CKPROP(dpool, \varphi) = valid

We must show that
(1) \( \forall B, \text{prog}, B \models \varphi \)

To show:
(2) \( \forall B, \forall \sigma_{p_1}, \ldots, \sigma_{p_n} \) for variables \( \vec{x}_1, \ldots, \vec{x}_n \) respectively, and \( \forall d_1, \ldots, d_n \),

\( \forall i \in [1, n] \), \text{prog}, \( B \models d_i; p_i(x_i^k)\sigma_{p_i} \)
and \( \models c_{p_i}(x_i^k, \ldots, x_n^k)\sigma_{p_i} \ldots \sigma_{p_n} \)
implies
\( \exists \sigma_{q_1}, \ldots, \sigma_{q_m} \) for variables \( \vec{y}_1, \ldots, \vec{y}_m \) respectively s.t.
\( \forall j \in [1, m] \), \text{prog}, \( B \models d_j; p_j(x_j^k)\sigma_{p_j} \)
and \( \models c_{p_j}(x_1^k, \ldots, x_n^k, y_j, \ldots, y_m)\sigma_{p_1} \ldots \sigma_{p_n} \sigma_{q_1} \ldots \sigma_{q_m} \).

Pick any \( B \),
pick any substitutions for arguments of \( p_1, \ldots, p_n \) of form \( \sigma_{p_1} = [t_1/x_1], \ldots, \sigma_{p_n} = [t_n/x_n] \),
and pick any \( d_1, \ldots, d_n \), s.t.
(3) \( \forall j \in [1, 2, \ldots, n] \), \text{prog}, \( B \models d_j; p_j(x_j^k)\sigma_{p_j} \)
and \( \models c_{p_j}(x_1^k, \ldots, x_n^k, y_j, \ldots, y_m)\sigma_{p_1} \ldots \sigma_{p_n} \sigma_{q_1} \ldots \sigma_{q_m} \).

(If there are no such substitutions, then the conclusion trivially holds.)

To show:
(4) \( \forall i \in [1, 2, \ldots, n] \),
either \( p_i \) is not on a cycle in \( G \) and
\( \exists (c_{p_i}(z_i^i), d_i(z_i^i):p_i(z_i)) \in \text{dpool}(p_i) \),
\( \exists \sigma_{d_i} \), s.t. \( c_{p_i}(z_i^i)\sigma_{d_i} \models z_i^i \),
\( \exists d_i'([z_i^i]:p_i(z_i)) \),
\( \exists \sigma_{d_i} \), s.t. \( \text{dpool} \vdash d_i([z_i^{i'}]:\sigma_{d_i}[z_i^i] \mapsto \gamma_{d_i|} d_i'(z_i^{i''}), \sigma_{c_i}^i) \),
and \( d_i = d_i'([z_i^{i''}], \sigma_{d_i}') \).
or \( p_i \) is on a cycle in \( G \) and
\( \exists (\text{rec},p_i(z_i^i), \Delta_{p_i}) \in \text{dpool}(p_i) \),
\( \exists \sigma_{d_i} \), s.t. \( \text{rec},p_i(z_i)\sigma_{d_i} \models z_i \),
\( \exists d_i'([z_i^{i''}]:p_i(z_i)) \),
\( \exists \sigma_{d_i} \), s.t. \( \text{dpool} \vdash (\text{rec}, p_i(z_i), \sigma_{d_i}[z_i] \mapsto \gamma_{d_i|} d_i'(z_i^{i''}), \sigma_{c_i}^i) \),
and \( d_i = d_i'([z_i^{i''}], \sigma_{d_i}') \).

Using (4),

72
By function \( \text{LOOKUpRec} \), (4), and (5),
(6) \( \forall i \in \{1, 2, \ldots, n\} \),
If \( p_i \) is on a cycle in \( G \),
\[ \exists (c_i(z_{c_i}'), d_i'(w_{c_i}')) \in \Delta p_i, \quad \exists \alpha \in \mathbb{Z}. \]
Every call to \( C \) where either
\[ c_i(z_{c_i}') \subseteq z_{c_i}'' \cup \alpha \]
implies
\[ \sigma_i''(z_{c_i'}') \subseteq \sigma_i''(z_{c_i'}''), \quad \alpha. \]

By the semantics of \( \text{GENDRULE, RNREC, RREC, RNREC} \), (4) and (6)
(7) \( \forall i \in \{1, 2, \ldots, n\} \),
\( z_{c_i}'' \) are fresh,
and either \( p_i \) is not on a cycle in \( G \) and \( z_{c_i}'' \subseteq z_{c_i}'' \),
or \( p_i \) is on a cycle in \( G \) and \( z_{c_i}'' \subseteq z_{c_i}'' \).

By the semantics of \( \text{LOOKUpRec} \) and \( \text{MERGEdERIVATION} \),
\( Z_1, \ldots, Z_n \) are variables in \( \text{dpool} \) which are arguments for \( p_1, \ldots, p_n \), corresponding to \( x_1, \ldots, x_n \),
where either \( p_i \) is on a cycle in \( G \) and \( Z_i = z_i \), or \( p_i \) is not on a cycle in \( G \) and \( Z_i = w_i \).
By the above, we can define the substitution
(8) \( \sigma = \bigcup_{i=1}^{n} [Z_i/z_i] \)

By assumption,
\( x_1, \ldots, x_n \) are mutually distinct variables
\( i \neq j \) implies \( \text{dom}([Z_i/z_i]) \cap \text{dom}([Z_j/z_j]) = \emptyset \)
Thus \( \sigma \) is well-defined

By (1) and the semantics of \( \text{CKPROP} \),
Every call to \( \text{CKPROP} \) in the \text{FOR} loop on Lines 10-13 of \text{CKPROP} returned “valid”
By the semantics of \( \text{LOOKUpRec, MERGEdERIVATION} \), (4) and (6)
Some \( \text{CKPROP} \) is called with the following arguments:
(9) \( c_i; \quad \bigwedge_{i=1}^{n} c_i'(Z_{c_i}') \)
where \( c_i'(Z_{c_i}') \) is a constraint for \( d_i'(Z_{d_i}') \),
and either \( p_i \) is on a cycle in \( G \) and \( c_i'(Z_{c_i}') = c_i'(z_{c_i}'') \), or \( p_i \) is not on a cycle in \( G \) and \( c_i'(Z_{c_i}') = c_i'(w_{c_i}') \).

\( c_{op} \): \( c_{op}(x_1, \ldots, x_n) = c_{op}(Z_1, \ldots, Z_n) \)
where \( c_{op}(x_1, \ldots, x_n) \) is the constraint for \( p_1, \ldots, p_n \) in \( \varphi \)
and every \( p_i \) is on a cycle in \( G \) and \( Z_i = z_i \), or \( p_i \) is not on a cycle in \( G \) and \( Z_i = w_i \).

\( d \): \( (d_i'(Z_{d_i}'))p_i(Z_i') :: \ldots :: (d_{n'}(Z_{d_{n'}})p_n(Z_n')) :: \emptyset \)
where \( d_i'(Z_{d_i}') \) are the derivations of \( p_i \) in \( \text{dpool} \) and
either \( p_i \) is on a cycle in \( G \) and \( Z_{d_i}' = z_{d_i}' \), or \( p_i \) is not on a cycle in \( G \) and \( Z_{d_i}' = w_{d_i}' \).

\( Q \): \( q_1(y_1), \ldots, q_m(y_m) \) are as they appear in \( \varphi \)
where \( c_{op}(x_1, \ldots, x_n, y_1, \ldots, y_m) \) is the constraint for \( q_1, \ldots, q_m \) in \( \varphi \)
and every \( p_i \) is on a cycle in \( G \) and \( Z_i = z_i \), or \( p_i \) is not on a cycle in \( G \) and \( Z_i = w_i \).

\( \text{CKPROP} \) returns “valid” under the following circumstances:
\text{Case A}: returns “valid” on Line 37
The true branch of the \text{IF}--\text{ELSE} statement from Lines 18-40 is taken,
The constraints for the derivations of \( p_1, \ldots, p_n \),
and the constraint for \( p_1, \ldots, p_n \) in \( \varphi \) are together satisfiable
The false branch of the \text{IF}--\text{ELSE} statement from Lines 21-37 is taken,
Every \( q_1, \ldots, q_m \) appears in some derivation of \( p_1, \ldots, p_n \)
The false branch of the \text{IF}--\text{ELSE} statement on Lines 34-37 is taken,
The constraints for the derivations of \( p_1, \ldots, p_n \),
the constraint for \( p_1, \ldots, p_n \) in \( \varphi \),
and the negation of the constraint for \( q_1, \ldots, q_m \) are together unsatisfiable

\text{Case B}: returns “valid” on Line 40
The false branch of the \text{IF}--\text{ELSE} statement from Lines 18-40 is taken,
The constraints for the derivations of \( p_1, \ldots, p_n \),
We derive a result that will be used in both the cases:

By (4) and (6),

\[ c_p(x_1',\ldots,x_n') [\sigma_1 \ldots \sigma_n] = c_p(x_1',\ldots,x_n') [t_1'/x_1] \ldots [t_n'/x_n] = c_p(t_1',\ldots,t_n') = c_p(Z_1',\ldots,Z_n') \Sigma_{d_1}' \ldots \Sigma_{d_n}' \]

where either \( p_i \) is not on a cycle in \( G \) and \( \Sigma_{d_i}' = \sigma_{d_i}' \),

or \( p_i \) is on a cycle in \( G \) and \( \Sigma_{d_i}'' = [z_{R_i}/(w_{d_i}' \backslash w_i)][z_i'/w_i] \sigma_{d_i}''. \)

By the above and (3),

\[ (10) = c_p(Z_1',\ldots,Z_n') \Sigma_{d_1}' \ldots \Sigma_{d_n}' \]

By (4) and (6),

\[ (11) = (\wedge_{i=1}^n c_i'(Z_{ei})') \Sigma_{d_1}' \ldots \Sigma_{d_n}' \]

By (10) and (11),

\[ (12) = (\wedge_{i=1}^n c_i'(Z_{ei})' \wedge c_p(Z_1',\ldots,Z_n')) \Sigma_{d_1}' \ldots \Sigma_{d_n}' \]

The false branch of the IF-ELSE statement on Lines 21-37 is taken, thus

(A.1) \( \forall j \) where \( j \in \{1, 2, \ldots, m\}, \)

\[ \exists i \in \{1, 2, \ldots, n\} \text{ s.t.} \]

\[ d_i'(Z_{d_i}') = (r_iD_i, p_i(Z_i), d_i(Z_{d_i}' \backslash Z_i)) \]

\[ \text{and } q_j \in d_i(Z_{d_i}' \backslash Z_i) \]

By (A.1) and the semantics of MERGELT,

(A.2) \( \forall q_{\ell'} \in \Sigma_q, \)

\[ \sigma_{q_{\ell'}} = \bigcup_{i=1}^n [Z_{i'} / y_{i'}] \]

where \( Z_{i'} \) is an argument for \( p_j \) as it appears in \( d_i(Z_{d_i}' \backslash Z_i) \)

The false branch of the IF-ELSE statement on Lines 34-37 is taken, thus

\[ \forall \sigma_a \in \neg (\wedge_{i=1}^n c_i'(Z_{ei})' \wedge c_p(Z_1',\ldots,Z_n')) \wedge \bigwedge_{i=1}^\lambda \neg c_q(Z_1',\ldots,Z_n, y_{i},\ldots,y_{m'}) \sigma_{q_i} \sigma_a \]

Which can be rewritten using De Morgan’s Laws as

\[ \forall \sigma_a \in (\neg (\wedge_{i=1}^n c_i'(Z_{ei})' \wedge c_p(Z_1',\ldots,Z_n')) \vee \bigvee_{i=1}^\lambda c_q(Z_1',\ldots,Z_n, y_{i},\ldots,y_{m'}) \sigma_{q_i} \sigma_a \]

Pick \( \sigma_a = \Sigma_{d_1}' \ldots \Sigma_{d_n}' \). Then

(A.3) \( \neg (\wedge_{i=1}^n c_i'(Z_{ei})' \wedge c_p(Z_1',\ldots,Z_n')) \Sigma_{d_1}' \ldots \Sigma_{d_n}' \vee (\bigvee_{i=1}^\lambda c_q(Z_1',\ldots,Z_n, y_{i},\ldots,y_{m'}) \sigma_{q_i} \Sigma_{d_1}' \ldots \Sigma_{d_n}' \]

By (A.3) and (12),

\[ (\bigvee_{i=1}^\lambda c_q(Z_1',\ldots,Z_n, y_{i},\ldots,y_{m'}) \sigma_{q_{\ell'}}) \Sigma_{d_1}' \ldots \Sigma_{d_n}' \]

Thus, there must be some \( \ell \in \{1, 2, \ldots, \lambda\} \) s.t.

(A.4) \( c_q(Z_1',\ldots,Z_n, y_{i},\ldots,y_{m'}) \sigma_{q_{\ell'}} \Sigma_{d_1}' \ldots \Sigma_{d_n}' \]

Using (7), (A.2), and (A.4), define

(A.5) \( \forall j \in \{1, 2, \ldots, m\}, \)

\[ \sigma_{q_j} = [Z_{i'} / y_{i'}] \Sigma_{d_1}' \ldots \Sigma_{d_n}' = [t_{i'} / y_{i'}] \]

where \( \exists i \in \{1, 2, \ldots, n\} \text{ s.t. } t_{i'} \subseteq \text{range}(\Sigma_{d_i}') \)

and \( \text{range}(\Sigma_{d_i}') = \text{range}(\sigma_{q_i}) \)

By (A.1), (A.2) and (A.4),

\[ \forall j \in \{1, 2, \ldots, m\} \exists i \in \{1, 2, \ldots, n\} \text{ s.t.} (d_i'(Z_{d_i}'):p_i(Z_i))::\text{nil} = \nu q_j(y_{i'}) \sigma_{q_{\ell'}} \]

Using (A.5), the above can be rewritten as

(A.6) \( \exists i \in \{1, 2, \ldots, n\} \text{ s.t.} (d_i'(Z_{d_i}'):p_i(Z_i))::\text{nil} = \nu q_j(Z_{i'}) \)

By (A.6), \( \forall j \in \{1, 2, \ldots, m\}, \)

(A.7) Either \( p_i \) is not on a cycle in \( G \),

and \((d_i'(z_{d_i}'):p_i(z_i))::\text{nil} = \nu q_j(z_{i'}) \)

By the above, (4), and Past Skeleton Derivations (Lemma 8),

\((d_i'(z_{d_i}'):p_i(z_i))::\text{nil} = \nu q_j(z_{i'}) \)

or \( p_i \) is on a cycle in \( G \),

and \((d_i'(w_{d_i}'):p_i(w_i))::\text{nil} = \nu q_j(w_{i'}) \)

Applying substitution \([z_{R_i}/(w_{d_i}' \backslash w_i)][z_i'/w_i] \) to the above expression, we have

\((d_i'(w_{d_i}'):z_{R_i}/(w_{d_i}' \backslash w_i))[z_i'/w_i]::\text{nil} = \nu q_j(z_{i'}) \)
By the above, (6), and Past Skeleton Derivations (Lemma 8),

\[(d''_n(z''_i):p_n(z''_i))::nil \models q_j(z''_j)\]

By (A.6) and (A.7),

(A.8) \(\forall j \in [1, m], (d''_1(z''_1):p_1(z''_1))::\cdots::(d''_n(z''_n):p_n(z''_n))::nil \models q_j(z''_j)\)

By applying \(\sigma''_{d1} \cdots \sigma''_{dn}\) throughout (A.8),

(A.7) \(\forall j \in \{1, 2, \ldots, m\}, (d''_1(z''_1):p_1(z''_1))::\cdots::(d''_n(z''_n):p_n(z''_n))::nil \models q_j(z''_j)\)

Using (4) and (A.5), this can be rewritten as

(A.7) \(\forall j \in [1, m], (d_1:p_1(x_1))::\cdots::(d_n:p_n(x_n))::nil \models q_j(y_j)\)

By (3), (4), (A.2), (A.5), and (A.7),

\[c^q(Z_1, \ldots, Z_n, y_1, \ldots, y_m)\sigma_{\text{false}} \Sigma''_d \cdots \Sigma''_m\]

\[= c^q(Z_1, \ldots, Z_n, y_1, \ldots, y_m)\prod_{i=1}^m [Z_i/y_j]\Sigma''_d \cdots \Sigma''_m\]

\[= c^q(Z_1, \ldots, Z_n, Z''_1, \ldots, Z''_m)\Sigma''_d \cdots \Sigma''_m\]

\[= c^q(x_1, \ldots, x_n, t_1, \ldots, t_m)[t_1/x_1] \cdots [t_m/x_n]\]

\[= c^q(x_1, \ldots, x_n, y_1, \ldots, y_m)\sigma_{p_1} \cdots \sigma_{p_n}\]

By the above and (A.4),

(A.7) \(\models c^q(x_1, \ldots, x_n, y_1, \ldots, y_m)\sigma_{p_1} \cdots \sigma_{p_n}\)

By (3), (A.5), (A.6) and (A.7),

the condition specified in (G) holds.

---

**Case B: CKPROP returns “valid” on Line 40**

The false branch of the IF-ELSE statement from Lines 18-40 is taken, thus

(B.1) \(\forall \sigma_p, \models \neg(\bigwedge_{i=1}^m c_i(Z_{ci}) \land c_p(Z_1, \ldots, Z_n))\sigma_p\)

(12) contradicts (B.1)