LaPeL: a Logic Programming Language

1 LaPeL Abstract Syntax

\[
\begin{align*}
x & \in \text{Variable} & f & \in \text{Function} & p & \in \text{Predicate} \\
t & \in \text{Term} ::= x | f(\vec{t}) \\
g & \in \text{Goal} ::= p(\vec{t}) | g_1 \land g_2 | g_1 \lor g_2 | a \supset g | \exists \vec{x}. g | \text{output} \\
a & \in \text{Assumption} ::= p(\vec{t}) | p(\vec{t}) \subset g | \forall \vec{x}. a \\
prog & \in \text{Program} ::= \vec{a} \vdash g
\end{align*}
\]

Where \( \vec{t} \) is a sequence of terms, \( \vec{x} \) is a sequence of variables, and \( \vec{a} \) is a sequence of assumptions. In the assumptions we write implication in the opposite direction than we normally do (i.e., “\( p(\vec{t}) \subset g \)” rather than “\( g \supset p(\vec{t}) \)”). This is a standard convention in logic programming (e.g., Prolog) because it makes it easier for the programmer to understand the program; the meaning of implication is unchanged. A program consists of an ordered sequence of assumptions followed by a goal (also known as the query). Think of the assumptions as the hypotheses in a judgement, and the query as the proposition that is supposed to be proved. The computation consists of attempting to prove the truth of the goal \( g \) given the hypotheses \( \vec{a} \).

The LaPeL syntax is an extension of definite Horn clauses; the added features are (1) the goal \( g_1 \lor g_2 \), which allows for disjunction; (2) the goal \( r \supset g \), which allows for hypothetical implication; and (3) the goal \text{output}, which outputs the values of all the variables currently in scope. These extensions preserve the primary benefit of definite Horn clauses: goal-directed execution, allowing for tractable computation—e.g., as with definite Horn clauses, the implications given in the assumptions \( \vec{a} \) have exactly one predicate as the succedent of the implication, thus when trying to prove a predicate goal \( p(\vec{t}) \) we know exactly which assumptions to apply.

2 Inference Rules

Below we give the inference rules for the subset of first-order logic defined in the syntax above, which we can use to prove the appropriate judgements. The \text{output} goal is treated as an axiom; its behavior of printing out the values of all the variables currently in scope is “extra-logical”, i.e., outside the scope of the logic. The inference rules for the conjunction, disjunction, existential quantification, and implication goals are all introduction rules (see the very first handout, on first-order logic and natural deduction). The reason is because those connectives are used in goals \( g \), i.e., we need to prove a proposition that contains those connectives—which is exactly what introduction rules are for. The inference rules for handling a predicate goal \( p(\vec{t}) \) make use of the elimination rules for universal quantification and implication, because we are using formulae that have universal quantifications and implications (i.e., the assumptions \( \vec{a} \)) to prove the goal \( p(\vec{t}) \)—which is exactly what elimination rules are for. The only difference from the inference rules given in handout 1 are the elimination rules for universal quantification, which we have specialized here because (1) the only universally quantified formulae we allow are atomic predicates and implications; and (2) we are only interested in conclusions involving atomic predicates.

As when we’re doing manual proofs using natural deduction, we’ll be applying these inference rules “bottom-up”. In other words, given a goal \( g \) that we’re trying to prove, we look for the appropriate inference rule that has that goal as its conclusion; in order to prove \( g \), then, we need to recursively prove the premises of that rule. One way to think about these inference rules is that we will use the introduction rules to decompose a goal into simpler sub-goals (by applying them bottom-up) until we get to some atomic predicate \( p(\vec{t}) \), and then use the elimination rules to prove that predicate using the assumptions \( \vec{a} \) (which may entail proving other goals using the same bottom-up process recursively).
Note that in the rules that substitute terms for variables (i.e., the \( \text{cons} \) can easily translate such notation into the functions order to get a realistic interpreter. For clarity in this example we will use Prolog’s list notation in the program and proof, though we ffi using an interpreter. In Section 3 we will explain the central problem causing us di

Example 1. We will trace through a proof of an example program using the given inference rules. It will become obvious based on this example that while the inference rules above correctly describe how to compute a solution to a given program, they do not immediately yield a practical computation strategy. That is, we do not yet have a tractable way to mechanically execute programs using an interpreter. In Section 3 we will explain the central problem causing us difficulty and how we can solve this problem in order to get a realistic interpreter. For clarity in this example we will use Prolog’s list notation in the program and proof, though we can easily translate such notation into the functions \text{cons} and \text{nil}. The example program is:

<table>
<thead>
<tr>
<th>Goals</th>
<th>Rules</th>
</tr>
</thead>
</table>
| \( p(\vec{r}) \) | \[
\frac{p(\vec{r}) \in \vec{d}}{\vec{d} \vdash p(\vec{r})} \quad (\text{Id})
\]
| \( p(\vec{r}) \subseteq g \in \vec{d} \) | \[
\frac{g \in \vec{d}}{\vec{d} \vdash p(\vec{r})} \quad (\supsetset)
\]
| \( g_1 \land g_2 \) | \[
\frac{\vec{d} \vdash g_1, \vec{d} \vdash g_2}{\vec{d} \vdash g_1 \land g_2} \quad (\land I)
\]
| \( g_1 \lor g_2 \) | \[
\frac{\vec{d} \vdash g_i, \ i \in [1, 2]}{\vec{d} \vdash g_1 \lor g_2} \quad (\lor I)
\]
| \( \exists x g \) | \[
\frac{\vec{d} \vdash g(\vec{x} \mapsto \vec{r})}{\vec{d} \vdash \exists x g} \quad (\exists I)
\]
| \( a \supset g \) | \[
\frac{a, \vec{d} \vdash g}{\vec{d} \vdash a \supset g} \quad (\supset I)
\]
| output | \[
\frac{\vec{d} \vdash \text{output}}{(\text{Out})}
\]

Note that in the rules that substitute terms for variables (i.e., the \( \exists I \) rule and the \( \forall E \) rules) the terms being substituted are ground terms (terms that do not contain any variables). Substituting a term that does contain variables wouldn’t make sense, because the point of the substitution is to map variables to actual values.

**Example 1.** We will trace through a proof of an example program using the given inference rules. It will become obvious based on this example that while the inference rules above correctly describe how to compute a solution to a given program, they do not immediately yield a practical computation strategy. That is, we do not yet have a tractable way to mechanically execute programs using an interpreter. In Section 3 we will explain the central problem causing us difficulty and how we can solve this problem in order to get a realistic interpreter. For clarity in this example we will use Prolog’s list notation in the program and proof, though we can easily translate such notation into the functions \text{cons} and \text{nil}. The example program is:

\[
\begin{align*}
A1. \ & \forall x. \text{eq}(x, x) \\
A2. \ & \text{iseven}(z) \\
A3. \ & \forall x. \text{iseven}(s(x)) \subseteq \text{isodd}(x) \\
A4. \ & \forall x. \text{isodd}(s(x)) \subseteq \text{iseven}(x) \\
A5. \ & \text{parity}([], []) \\
A6. \ & \forall wxyz. \text{parity}([x | y], [w | z]) \subseteq \text{eq}(w, \text{even}) \land \text{iseven}(x) \land \text{parity}(y, z) \\
A7. \ & \forall wxyz. \text{parity}([x | y], [w | z]) \subseteq \text{eq}(w, \text{odd}) \land \text{isodd}(x) \land \text{parity}(y, z) \\
Q. \ & \exists xy. \text{parity}([x | y], [\text{odd}, \text{even}]) \land \text{output}
\end{align*}
\]

Let \( \vec{d} \) be the assumptions \( A1 \sim A7 \); then the judgement that we need to prove is \( \vec{d} \vdash \exists xy. \text{parity}([x | y], [\text{odd}, \text{even}]) \land \text{output} \). A computation in logic programming is a proof search, i.e., program execution consists of trying to find a proof of the given judgement. In this case, a proof consists of finding two numbers, the first of which is odd and the second of which is even. One proof (out of an infinite number of possible proofs) is shown below; this computation would output \( x \mapsto s(z) \) and \( y \mapsto z \).
In general we want our logic program interpreter to output all possible solutions; given such an interpreter it is trivial to modify it to only output the first solution it finds if that’s what we want. This means that our proof search shouldn’t stop at the first proof it finds, but continue on to try and find as many proofs as it can. Because, as in this case, there may be an infinite number of solutions, the interpreter needs to output the solutions as it finds them rather than waiting until it has computed them all and outputting the solutions all at once.

3 Handling Nondeterminism

A nondeterministic computation is one that can take a variety of routes to get to a solution and that can potentially arrive at multiple different solutions. An example of nondeterminism that you have seen before are nondeterministic finite automata (NFA), where from a single state we could transition to multiple next states on the same input. During a nondeterministic computation there are choice points where the computation can go in different directions (as, for example, an NFA can choose which next state to go to).

A computation succeeds if there is at least one choice it can make that leads to a valid solution (as, for example, an NFA accepts an input if there is at least one choice for next state that leads to a valid final state).

Logic programming is inherently nondeterministic. In fact, we can view logic programming essentially as a search problem, because computing solutions means trying to find the right choices during computation that will eventually lead to a solution. We can categorize the kinds of choices the computation needs to make:

- **Term Selection:** When applying the $\exists$ rule, the interpreter needs to choose terms to substitute for the variables being existentially quantified. In other words, in the premise of the $\exists$ rule for the goal $\exists x.r$, we have to select a sequence of terms $t_x$ that we will substitute for the sequence of variables $x_i$ to create a new goal $g[x_i \mapsto t_x]$. In the example, we choose terms $s(z)$ and $z$ to substitute for variables $x$ and $y$, respectively. There are actually an infinite number of terms that the interpreter could choose—an infinite number of which would lead to correct solutions and an infinite number of which would lead to failure.

- **Assumption Selection:** When attempting to prove an atomic predicate goal $p(r)$, the interpreter must choose which assumption to use. In some cases the choice is obvious (e.g., in the example when trying to prove the predicate isodd there is only one applicable assumption), but in other cases it isn’t obvious (e.g., in the same example when trying to prove parity there are multiple applicable assumptions and it isn’t clear which one will work until we try to prove the resulting goals).

- **Conjunct Ordering:** When applying the conjunction rule, which has multiple judgements as premises, the interpreter must choose which judgement premise to attempt to prove first. While logically conjunction is commutative and ordering shouldn’t matter, it turns out that because we’re applying proof search it does matter, as demonstrated below.

- **Disjunct Ordering:** When applying the disjunction rule, which has one of two possible judgement premises, the interpreter must choose which judgement premise to attempt to prove. Just as with conjunction, while disjunction is logically commutative the ordering does still matter because of our proof search computation, as demonstrated below.

The reason that the inference rules alone are not sufficient to yield a practical logic programming interpreter is that those rules don’t give the interpreter any guidance on how to make those choices. We will take each category of choice in turn, clarify the problem that choice represents, and then describe a method to guide the interpreter in making that choice, thus yielding a practical interpreter.
3.1 Term Selection

When applying the $\exists I$ rule, the interpreter must choose a sequence of terms to substitute for the existentially quantified variables. However, there may be an infinite set of terms to choose from, and the rule provides no guidance on which terms to choose. In the example from Section 2, there are an infinite number of possibilities that will cause the proof to fail; consistently bad choices by the interpreter can lead to nontermination. To get a tractable computation, we somehow have to know which terms will work before actually performing the computation to see which terms will work—i.e., we need to be able to see into the future. Fortunately, there is a way that we can “cheat” using two new notions: logic variables and unification. Using these notions, we can defer our choice of terms until later in the computation where we may have a better idea of which terms we should choose.

3.1.1 Logic Variables.

A logic variable $X$ represents some specific ground term (i.e., a term that does not contain any variables), but we are deferring the choice of which term it represents. Do not be confused by the terminology here: variables (that is, $x \in \text{Variable}$) are syntactic terms in our language, typed in by the programmer, while logic variables (that is, $X \in \text{LogicV}$) are values generated by the language runtime while executing a program in order to defer choosing specific ground terms when employing the $\exists I$ rule. Whenever the runtime employs the $\exists I$ rule and needs to choose a sequence of ground terms $\vec{t}$, it generates a sequence of fresh logic variables $\vec{X}$ and uses them instead. In the Section 2 example, we would have:

\[
\begin{align*}
\vec{d} & \vdash \text{parity}(X, Y), [\text{odd, even}] \land \text{output} \\
\vec{d} & \vdash \exists X \text{parity}(x, y), [\text{odd, even}] \land \text{output}
\end{align*}
\]

In this version of the proof logic variables $X$ and $Y$ represent that some ground terms have been chosen, but they don’t immediately commit to a specific choice. The proof will act on the logic variables just as if they were terms up until the point when it absolutely needs to know what terms they really stand for. By then, it will know what terms they need to be in order for the proof to potentially succeed, and it can instantiate the logic variable to the desired terms using a process called unification.

At some point in the computation, we need to commit to a decision and map logic variables to actual terms. If we look at the inference rules, the point where we absolutely need to know what term a logic variable represents is in the elimination rules for trying to prove an atomic predicate goal—the interpreter needs to check whether the terms used in the predicate goal match the terms used in the atomic predicate assumption (or in the head predicate of the clause assumption, depending on which elimination rule we’re using). Given an assumption $a$ that we’re trying to use to prove an atomic predicate goal $g$, our strategy is to attempt to map the logic variables used in goal $g$ to terms that will allow assumption $a$ to match. The idea is that at this point in the computation, we can say “if we had chosen these specific terms when applying the $\exists I$ rule, then when the computation reached this point we would have matched assumption $a$ and been able to prove goal $g$”. Of course, we must be consistent in how we map the logic variables; if we mapped a logic variable $X$ to some term $t_1$ to prove one goal, we can’t then map $X$ to some other term $t_2$ to prove a different goal.

Example 2. Consider the following example program:

\[
\begin{align*}
A1. & \quad \text{foo}(42) \\
A2. & \quad \text{bar}(42) \\
Q. & \quad \exists X \text{foo}(x) \land \text{bar}(x)
\end{align*}
\]

The proof for this program (there is only one valid proof) would look like the following:

\[
\begin{align*}
\vec{d} & \vdash \text{foo}(X) \quad \text{(Id, where } X = 42) \\
\vec{d} & \vdash \text{bar}(X) \quad \text{(Id, where } X = 42) \\
\vec{d} & \vdash \text{foo}(X) \land \text{bar}(X) \quad \text{(\&I)} \\
\vec{d} & \vdash \exists X \text{foo}(x) \land \text{bar}(x) \quad \text{($\exists I$)}
\end{align*}
\]

When applying the $\exists I$ rule we substituted the logic variable $X$ for the variable $x$. When attempting to prove the atomic predicate goal $\text{foo}(X)$, the only way that we could find a matching assumption is if $X = 42$. Similarly, when trying to prove $\text{bar}(X)$ the only way we could find a matching assumption is if $X = 42$. Therefore the proof succeeds and the solution for the query is that $x = 42$. However, consider the following slightly modified program:
The proof for this program (again there is only one) would look like the following:

Here is a different example that uses multiple logic variables:

A1. ∀x.eq(x, x)
A2. ∀x.build(x, bar(x))
A3. foo(42)
Q. ∃xyz.eq(y, z) ∧ build(x, y) ∧ foo(x)

The proof for this program (again there is only one) would look like the following:

From attempting to prove eq(Y, Z) we know that Y and Z must be the same; from attempting to prove build(X, Y) we know that Y must be the same as bar(X); and from attempting to prove foo(X) we know that X must be 42. Hence the solution to the query is x = 42, y = z = bar(42). We can see that to achieve these results we need two things: (1) some way to keep track of the equivalences between logic variables and terms (e.g., Y = Z = bar(X) and X = 42); and (2) some way to perform matching between predicates in order to figure out what the equivalences should be. These issues are discussed in the following section.

Implementation Sidenote: The ∃I inference rule technically uses substitution, i.e., it will search through the goal g to find all instances of the variable x and replace them with the new term (or logic variable). For the same reasons as when we described lambda calculus, we will be using environments, which in this case will be a mapping from variables to logic variables. We are already pairing a goal with the assumptions we can use to prove that goal (i.e., d ⊢ g); we extend this pairing to include an environment for the goal and call the result a closure. In Example 1, e.g., rather than doing substitution and changing “d ⊢ parity([x, y], [odd, even])” to “d ⊢ parity([X, Y], [odd, even])” we will compute the closure “d ⊢ (parity([x, y], [odd, even]), [x ↦ X, y ↦ Y]).”

3.1.2 Unification.

The constraints on logic variables that we gather while attempting to build a proof (e.g., the constraints on X, Y, and Z in Example 3 above) are all equality constraints—they specify equalities among logic variables (e.g., Y = Z) and between logic variables and terms (e.g., Y = bar(X) and X = 42). The set of equality constraints that we gather for a specific proof defines an equivalence relation. An equivalence relation partitions a set into disjoint subsets (called equivalence classes) such that every element in the same subset is equivalent. In Example 3, the equivalence relation partitions the set of logic variables and terms into the following two subsets:

• { Y, Z, bar(X), bar(42) }
• { X, 42 }

However, we can’t specify arbitrary equalities. For example, the second program from Example 2 above would require the following equivalence relation:

• { X, 12, 42 }
But clearly 12 and 42 are not equivalent—they are completely different ground terms. Therefore we need a way to determine when we add an equality constraint whether that constraint makes sense—that is, whether adding that constraint would put non-equivalent things into the same equivalence class. If it does make sense then we continue building the proof; if it doesn’t make sense then the proof fails. The method we use to determine whether an equality constraint makes sense is called unification.

**Implementation Sidenote:** We will need a data structure to store the equivalence relation as we are building it. Because we’re trying to keep track of constraints on logic variables we don’t actually care about equivalences between terms; only equivalences that involve logic variables. There are many different data structures that we could use; for our implementation we’ll keep it simple and use a map from logic variables to values, where values are either logic variables or terms. For example, the equivalence relation from Example 3 would look like:

\[
\begin{align*}
X & \rightarrow 42 \\
Y & \rightarrow \text{bar}(X) \\
Z & \rightarrow Y
\end{align*}
\]

If we need the value of a logic variable, we look it up in the map. If it maps to another logic variable, we recursively look it up in the map, and so on until we get to something that isn’t a logic variable or we get to a logic variable that doesn’t have an entry in the map. For example, if we look up the value of Z we get Y; looking up Y we then get \text{bar}(X). If we need the complete ground term, then we also need to look up the value of any logic variables used inside the resulting term—for example, given \text{bar}(X) we could look up X to get 42, and hence the complete ground term is \text{bar}(42). Because we build the equivalence relation as we build a proof, it is possible that an equivalence class does not yet contain a term, i.e., that it only has logic variables. In this case, looking up logic variables recursively would terminate when the logic variable we’re looking up doesn’t have an entry in the map. The value that the recursive map lookup terminates with (whether it be a logic variable or a term) is called the set representative of the equivalence class.

**Unification Algorithm.** Unification is a method of determining whether two values can possibly be equivalent or not. The effect of trying to unify two values is either (1) success, with an updated equivalence relation; or (2) failure. We can represent these possibilities with an Option type, where None represents failure. The unification procedure pseudocode is below, where \(v_1, v_2\) are values (i.e., logic variables or terms), eq is a data structure representing the current equivalence relation, and eq(v) looks up the value \(v\) in the equivalence relation eq to get its set representative.

```haskell
unify(v1, v2, eq) =
    let rep1 = eq(v1), rep2 = eq(v2) in
    match (Some(eq), Some(eq[X \mapsto rep2]), Some(eq[X \mapsto rep1]), unifyTerms(v1, v2, eq), None) with
    { if rep1 = rep2 then Some(eq) |
        if rep1 = X and X does not occur in rep2 then Some(eq[X \mapsto rep2]) |
        if rep2 = X and X does not occur in rep1 then Some(eq[X \mapsto rep1]) |
        if rep1 = f(v1), rep2 = f(v2), |v1| = |v2| then Some(newEq) |
        None
    }

unifyTerms(pairs, eq) = pairs.foldLeft(Some(eq))(
    (acc, (v1, v2)) \mapsto acc match {
        case None \Rightarrow None
        case Some(newEq) \Rightarrow unify(v1, v2, newEq)
    }
)
```

In unify, the first thing we do is get the set representatives of the values \(v_1\) and \(v_2\). Then we take one of five actions depending on certain conditions:

1. If the two set representative are identical it means that \(v_1\) and \(v_2\) are already equivalent, so we succeed and return the same equivalence relation we started with.
2. Otherwise, if \( rep_1 \) is a logic variable \( X \) that does not occur in \( rep_2 \) then we put \( X \) in the same equivalence class as \( rep_2 \) by returning a modified equivalence relation that maps \( X \) to \( rep_2 \). We have to have the extra condition that \( X \) does not occur in \( rep_2 \) in order to prevent unification from creating an infinite-size term (which is not legal). For example, if we make \( X \) and \( \text{bar}(X) \) equivalent then the complete ground term for \( X \) is an infinitely nested sequence of calls \( \text{bar} \left( \text{bar}(\cdots) \right) \).

3. Otherwise, if \( rep_2 \) is a logic variable \( X \) that does not occur in \( rep_1 \) then we put \( X \) in the same equivalence class as \( rep_1 \) by returning a modified equivalence relation that maps \( X \) to \( rep_1 \).

4. Otherwise, if \( rep_1 \) is a function call to function \( f \) and \( rep_2 \) is a function call to the same function \( f \) and both calls have the same number of arguments, then we delegate to \( \text{unifyTerms} \).

5. Otherwise, unification fails (i.e., the given values \( v_1 \) and \( v_2 \) cannot possibly be equivalent) and we return \text{None}.

The helper function \( \text{unifyTerms} \) just iterates through the arguments of the two functions and tries to unify the terms that are in the same argument position by making a recursive call to \( \text{unify} \). If any of the arguments cannot be unified then \( \text{unifyTerms} \) will return \text{None}, otherwise it will return the updated equivalence relation.

**Using Unification.** When we’re trying to compute a solution to a program, we keep track of the current equivalence relation. Whenever we attempt to prove an atomic predicate goal by matching it with an assumption, we update the equivalence relation accordingly using unification. If the unification succeeds then we continue on with the proof using the updated relation; if the unification fails then we know this proof isn’t valid and we try to find a different proof.

### 3.1.3 Proof Search

Recall that for logic programming computation is proof search, and that we want to use proof search to find all possible solutions for a given program. How does that fit in with using equivalence relations and unification? The following example will help illustrate.

**Example 4.** Consider the following program:

\[
\begin{align*}
A1. \quad \text{foo}(12) \\
A2. \quad \text{foo}(42) \\
A3. \quad \forall x. \text{eq}(x, x) \\
Q. \quad \exists x y. \text{eq}(x, y) \land \text{foo}(x)
\end{align*}
\]

We can see by inspection that there are two possible solutions: \( x = y = 12 \) and \( x = y = 42 \). The structure of the proof for both solutions looks like this:

\[
\frac{d \vdash \text{id}(X, Y)}{d \vdash \text{eq}(X, Y)} \quad \forall \text{id} \text{ where } X = Y \quad \frac{d \vdash \text{foo}(X)}{d \vdash \text{foo}(X)} \quad \text{Id where } X = ?
\]

\[
\frac{d \vdash \text{eq}(X, Y) \land \text{foo}(X)}{d \vdash \exists x y. \text{eq}(x, y) \land \text{foo}(x)} \quad \exists \text{I}
\]

The difference in the two solutions is in the proof of \( \text{foo}(X) \): one proof uses assumption \( A1 \) and so \( X = 12 \), the other proof uses \( A2 \) and so \( X = 42 \). The way that we find both solutions is to try all applicable assumptions when attempting to prove an atomic predicate goal (e.g., \( \text{foo}(X) \)). When we reach the part of the computation attempting to prove an atomic predicate goal where the current equivalence relation is \( \text{eq} \), we iterate through all of the assumptions and try to complete the proof using each one in turn, using the original equivalence relation \( \text{eq} \) for each attempt. In example 4 above:

1. When we apply the \( \exists \text{I} \) rule we introduce the logic variables \( X \) and \( Y \). They each start out being equivalent only to themselves, so they are each set representatives. The equivalence relation \( \text{eq} = [] \) (the empty map).

2. When we attempt to prove \( \text{eq}(X, Y) \) there is only one assumption that applies, \( A3 \). Matching the goal to the assumption updates the equivalence relation so that \( \text{eq} = [X \mapsto Y] \).

3. When we attempt to prove \( \text{foo}(X) \) there are two assumptions that apply, \( A1 \) and \( A2 \).

   (a) We first try to match with \( A1 \) using \( \text{eq} = [X \mapsto Y] \). This match is successful and updates the equivalence relation so that \( \text{eq} = [X \mapsto Y, Y \mapsto 12] \). That fulfills the proof and we have our first solution.
The first part of the proof search would look like this (where reach the base case, hence the recursion will never terminate and no solutions will ever be output. foo is then again and the interpreter again selected term recursively. Suppose for one proof the interpreter selects foo the assumption selection strategy controls the search strategy used to find the proofs. Example 6. Consider the following program:

Here is a contrived but simple example that shows conjunct ordering matters.

### 3.2 Assumption Selection

Here is a simple example that shows the order in which we try assumptions matters.

**Example 5.** Consider the following program:

\[
A1. \text{build}(\text{foo}(\text{nil})) \\
A2. \forall x . \text{build}(\text{foo}(x)) \subset \text{build}(x) \\
Q. \exists x . \text{build}(x) \wedge \text{output}
\]

This program specifies a \text{foo} term with an arbitrarily number of nested \text{foo} terms inside of it. For this program there are an infinite number of correct solutions (i.e, \text{foo}(\text{nil}), \text{foo}(\text{foo}(\text{nil})), etc). The issue here is proving a \text{build} predicate; specifically, which assumption should the interpreter select? This choice controls which solution the interpreter is going to compute. The interpreter should compute all the solutions (or as many as we give it time for if there are an infinite number), but the assumption selection strategy controls the order in which the solutions are computed. Remember that for logic programming, computation is proof search; the assumption selection strategy controls the search strategy used to find the proofs.

In this example assumption A1 is the base case, it terminates the recursion, and assumption A2 is the inductive case, it builds a \text{foo} term recursively. Suppose for one proof the interpreter selects A2 to prove the query \text{build}(x), which recursively calls \text{build} again and the interpreter again selected A2, which recursively calls \text{build} again and the interpreter selects A1. The computed solution is then \text{foo}(\text{foo}(\text{foo}(\text{nil}))). For the next proof perhaps the interpreter immediately selects A1 to prove the query, and the computed solution is then \text{foo}(\text{nil}). In the worst case, suppose the interpreter always chooses A2 when proving \text{build}—then it will never reach the base case, hence the recursion will never terminate and no solutions will ever be output.

The intent of the programmer is for this program to output an infinite series of \text{foo} terms \text{foo}(\text{nil}), \text{foo}(\text{foo}(\text{nil})), etc. The central problem is that the nondeterminism inherent in assumption selection doesn’t give the programmer any control over how the interpreter behaves, and so the programmer doesn’t have any guarantees about what order things will be proven in or even if the computation will ever terminate.

**Solution.** We resolve this problem by enforcing that the interpreter must select the assumptions it uses to try and prove a predicate goal in a top-down order. That is, given a predicate goal \( p \), the interpreter must go through the assumptions in order starting from A1. Remember that for a particular predicate goal \( p \), each different assumption selected to prove that goal is yielding a different proof; by enforcing the order in which assumptions are selected we are enforcing an ordering on proofs. This solution gives the programmer control—because they know the interpreter will select the assumptions top-down, they can control how the interpreter behaves by putting the assumptions in a particular order that will generate solutions in the desired order.

### 3.3 Conjunct Ordering

Here is a contrived but simple example that shows conjunct ordering matters.

**Example 6.** Consider the following program:

\[
A1. \forall x . \text{eq}(x, x) \\
A2. \text{loop} \subset \text{loop} \\
Q. \text{eq}(\text{foo}, \text{bar}) \wedge \text{loop}
\]

The first part of the proof search would look like this (where \( d = A1–A2 \):
If the interpreter chooses to solve the premise \( \text{eq}(\text{foo}, \text{bar}) \) first, then it will immediately find that the proof is guaranteed to fail and there does not exist any proof, and so the interpreter will terminate with failure. However, if the interpreter chooses to solve the premise \( \text{loop} \) first, then it will attempt to prove \( \text{loop} \) by proving \( \text{loop} \), which leads to an infinite recursion and the interpreter will never terminate at all. Here is a less contrived example that shows a similar problem:

**Example 7.** Consider the following program, where for convenience we’ll assume integers (i.e., -1, 0, 1, etc), the predicate on integers \( \text{gtz} \) (standing for “greater-than-zero”), and the predicate on integers \( \text{pred} \) (relating a number to its predecessor) are defined in some set of assumptions not shown here:

- \( A1. \) \text{build}(\text{foo}(\text{nil}), 0) \\
- \( A2. \) \( \forall xy. \text{build}(\text{foo}(x), y) \subset \exists z. \text{gtz}(y) \land \text{pred}(y, z) \land \text{build}(x, z) \) \\
- \( Q. \) \( \exists x. \text{build}(x, 1) \land \text{output} \)

This program will compute a nested series of calls to the function \( \text{foo} \) where the number of nested calls is equal to the number given in the query—in this case, one, so the output should be \( x \mapsto \text{foo}(\text{foo}(\text{nil})) \). However, the order that the interpreter proves these goals again matters for termination. Specifically, the issue is the body of the clause for \( \text{build} \): the goal \( \text{gtz}(y) \) is a boundary check that is intended to stop the recursion when the \( y \) argument gets to 0, but if the interpreter decides to prove the goal \( \text{build}(x, z) \) first, before making that check, then the recursion will never terminate.

**Solution.** We resolve this problem by enforcing that the interpreter must attempt to prove conjuncted goals left-to-right, i.e., it must prove the leftmost goal first and the rightmost goal last; if any of those conjuncted goals fails then none of the goals after it are attempted. This solution gives the programmer control—because they know the interpreter will attempt the goals left-to-right, they can control how the interpreter will behave by putting the conjuncted goals in a particular order that has the behavior they want.

**Special Note.** While the assumption selection problem and solution look very similar to the conjunct ordering problem and solution, they are fundamentally addressing different things. (1) Conjunct ordering specifies within a particular proof what order the premises of the \( \land I \) rule are proven. By enforcing a left-to-right ordering on conjuncts the programmer can reliably enforce that a particular proof must be finite, and hence the computation of that proof must terminate (with either success or failure). (2) Assumption selection specifies the order in which proofs are generated during proof search. By enforcing a top-down ordering on assumptions the programmer can reliably control what order proofs will be generated in.

### 3.4 Disjunct Ordering

The issue of disjunct ordering is more closely related to assumption selection than it is to conjunct ordering. Depending on which premise of the \( \lor I \) rule it chooses to prove, the interpreter will find different proofs. Because we want the interpreter to explore all possible proofs it will need to explore both choices, but the order in which it explores them will control the order in which it finds the solutions. In order to give the programmer control over the proof search strategy (and thus the order in which proofs are explored and solutions are found), we need to give the programmer control over the order in which the disjuncts are proven.

**Solution.** We resolve this problem by enforcing that the interpreter must attempt to prove disjuncted goals left-to-right, i.e., it must prove the leftmost goal first and the rightmost goal last. Unlike conjuncted goals, disjuncted goals are independent—the interpreter will try to prove each of them regardless of whether the previous one failed or succeeded (in order to explore all possible proofs).

### 4 Example Redux

Here is a trace of the computation for the program in Example 1. The tree derivation format we used previously works well for displaying single proofs but not for displaying multiple proofs branching off of each other, so we’ll resort to a more ad-hoc format for displaying the proof search. The state of the computation will be a stack of closures that we need to resolve and a data structure representing the current equivalence relation. For brevity we don’t list the assumptions \( \bar{a} \) in the closures; since we aren’t using hypothetical implication in this example they are always the same, i.e., \( A1 \ldots A7 \). Here is the program from the example:
A1. ∀x.eq(x, x)
A2. iseven(z)
A3. ∀x.iseven(s(x)) ⊆ isodd(x)
A4. ∀x.isodd(s(x)) ⊆ iseven(x)
A5. parity([], [])
A6. ∀wxyz.parity([x], [w | z]) ⊆ eq(w, even) ∧ iseven(x) ∧ parity(y, z)
A7. ∀wxyz.parity([x], [w | z]) ⊆ eq(w, odd) ∧ isodd(x) ∧ parity(y, z)
Q. ∃xy.parity([x], [odd, even]) ∧ output

Here is the trace of the computation:

1. Starting state.

<table>
<thead>
<tr>
<th>Goal</th>
<th>Environment</th>
<th>Equivalence Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>∃xy.parity([x], [odd, even]) ∧ output</td>
<td>[]</td>
<td>[]</td>
</tr>
</tbody>
</table>

2. ∃I: Create logic variables X1 and Y1 and create new goal closure.

<table>
<thead>
<tr>
<th>Goal</th>
<th>Environment</th>
<th>Equivalence Relation</th>
</tr>
</thead>
</table>
| parity([x], [odd, even]) ∧ output | x ↦ X1  
y ↦ Y1 | []                  |

3. ∧I: Separate the conjunction into two separate goals.

<table>
<thead>
<tr>
<th>Goal</th>
<th>Environment</th>
<th>Equivalence Relation</th>
</tr>
</thead>
</table>
| parity([x], [odd, even])  | x ↦ X1      
y ↦ Y1    | []                  |
| output                    |             |                     |

4. Attempt to prove atomic predicate goal on top of goal stack.


[X1, Y1] ≠ []     [odd, even] ≠ []

Unification failure: backtrack.

4.b. ∀Ez: Try assumption A6, creating logic variables W1, X2, Y2, Z1.

W1 ≡ odd     X2 ≡ X1     Y2 ≡ [Y1]     Z1 ≡ [even]

<table>
<thead>
<tr>
<th>Goal</th>
<th>Environment</th>
<th>Equivalence Relation</th>
</tr>
</thead>
</table>
| eq(w, even) ∧ iseven(x) ∧ parity(y, z) | w ↦ W1 
x ↦ X2  
y ↦ Y2  
z ↦ Z1 | []
| output                    | x ↦ X1  
y ↦ Y1   | [W1 ↦ odd 
X1 ↦ X2  
Y2 ↦ [Y1]  
Z1 ↦ [even]]

4.b.i. ∧I: Separate the conjunction into two separate goals.
Goal Environment Equivalence Relation

\[
\begin{aligned}
\text{eq}(w, \text{even}) & \quad W_1 \mapsto \text{odd} \\
\text{iseven}(x) \land \text{parity}(y, z) & \quad \begin{align*}
W_2 \mapsto & X_1 \\
x \mapsto & X_4 \\
y \mapsto & Y_3 \\
z \mapsto & Z_2
\end{align*} & \quad \begin{align*}
Y_3 \mapsto & [Y_1] \\
Z_2 \mapsto & \text{even}
\end{align*}
\end{aligned}
\]

4.b.ii. Attempt to prove atomic predicate goal on top of goal stack.

4.b.ii.A. ∃E\text{Id}: Try assumption A1, creating logic variable X_1.

odd \neq \text{even}

Unification failure: backtrack.

4.c. ∀E\text{a}: Try assumption A7, creating logic variables W_2, X_4, Y_3, Z_2.

\[
\begin{aligned}
W_2 \equiv & \text{odd} \\
X_4 \equiv & X_1 \\
Y_3 \equiv & [Y_1] \\
Z_2 \equiv & \text{even}
\end{aligned}
\]

4.c.i. ∧a: Separate the conjunction into two separate goals.

Goal Environment Equivalence Relation

\[
\begin{aligned}
\text{eq}(w, \text{odd}) \land \text{isodd}(x) \land \text{parity}(y, z) & \quad W_2 \mapsto \text{odd} \\
\text{isodd}(x) \land \text{parity}(y, z) & \quad \begin{align*}
W_2 \mapsto & X_1 \\
x \mapsto & X_4 \\
y \mapsto & Y_3 \\
z \mapsto & Z_2
\end{align*} & \quad \begin{align*}
X_4 \mapsto & X_1 \\
Y_3 \mapsto & [Y_1] \\
Z_2 \mapsto & \text{even}
\end{align*}
\end{aligned}
\]

4.c.ii. Attempt to prove atomic predicate goal on top of goal stack.

4.c.ii.A. ∀E\text{Id}: Try assumption A1, creating logic variable X_5.

\[X_5 \equiv \text{odd}\]

Success! Continue with goal stack.

Goal Environment Equivalence Relation

\[
\begin{aligned}
\text{isodd}(x) \land \text{parity}(y, z) & \quad W_2 \mapsto \text{odd} \\
\text{output} & \quad \begin{align*}
X_1 \mapsto & X_4 \\
Y_3 \mapsto & [Y_1] \\
Z_2 \mapsto & \text{even}
\end{align*}
\end{aligned}
\]

4.c.ii.B. ∧a: Separate the conjunction into two separate goals.
### Goal Environment Equivalence Relation

<table>
<thead>
<tr>
<th>Goal</th>
<th>Environment</th>
<th>Equivalence Relation</th>
</tr>
</thead>
</table>
| isodd(x) | \[
  \begin{array}{l}
  w \mapsto W_2 \\
  x \mapsto X_4 \\
  y \mapsto Y_3 \\
  z \mapsto Z_2 \\
  \end{array}
  \] | \[
  \begin{array}{l}
  W_2 \mapsto \text{odd} \\
  X_1 \mapsto X_4 \\
  X_5 \mapsto \text{odd} \\
  \end{array}
  \] |
| parity(y, z) | \[
  \begin{array}{l}
  w \mapsto W_2 \\
  x \mapsto X_4 \\
  y \mapsto Y_3 \\
  z \mapsto Z_2 \\
  \end{array}
  \] | \[
  \begin{array}{l}
  Y_3 \mapsto [Y_1] \\
  Z_2 \mapsto [\text{even}] \\
  \end{array}
  \] |
| output | \[
  \begin{array}{l}
  x \mapsto X_1 \\
  y \mapsto Y_1 \\
  \end{array}
  \] | \[
  \begin{array}{l}
  \end{array}
  \] |

4.c.ii.C. Attempt to prove atomic predicate goal on top of goal stack.

4.c.ii.I. \( \forall E : \) Try assumption A4, creating logic variable \( X_6 \).
\( X_4 \equiv s(X_6) \)

<table>
<thead>
<tr>
<th>Goal</th>
<th>Environment</th>
<th>Equivalence Relation</th>
</tr>
</thead>
</table>
| iseven(x) | \[
  \begin{array}{l}
  x \mapsto X_6 \\
  \end{array}
  \] | \[
  \begin{array}{l}
  W_2 \mapsto \text{odd} \\
  X_1 \mapsto X_4 \\
  X_5 \mapsto \text{odd} \\
  \end{array}
  \] |
| parity(y, z) | \[
  \begin{array}{l}
  w \mapsto W_2 \\
  x \mapsto X_4 \\
  y \mapsto Y_3 \\
  z \mapsto Z_2 \\
  \end{array}
  \] | \[
  \begin{array}{l}
  Y_3 \mapsto [Y_1] \\
  Z_2 \mapsto [\text{even}] \\
  \end{array}
  \] |
| output | \[
  \begin{array}{l}
  x \mapsto X_1 \\
  y \mapsto Y_1 \\
  \end{array}
  \] | \[
  \begin{array}{l}
  \end{array}
  \] |

4.c.ii.C.II. Attempt to prove atomic predicate goal on top of goal stack.

4.c.ii.C.II.1. \( \forall E \Delta : \) Try assumption A2.
\( X_6 \equiv z \)
Success! Continue with goal stack.

<table>
<thead>
<tr>
<th>Goal</th>
<th>Environment</th>
<th>Equivalence Relation</th>
</tr>
</thead>
</table>
| parity(y, z) | \[
  \begin{array}{l}
  w \mapsto W_2 \\
  x \mapsto X_4 \\
  y \mapsto Y_3 \\
  z \mapsto Z_2 \\
  \end{array}
  \] | \[
  \begin{array}{l}
  W_2 \mapsto \text{odd} \\
  X_1 \mapsto X_4 \\
  X_5 \mapsto \text{odd} \\
  \end{array}
  \] |
| output | \[
  \begin{array}{l}
  x \mapsto X_1 \\
  y \mapsto Y_1 \\
  \end{array}
  \] | \[
  \begin{array}{l}
  X_6 \mapsto z \\
  Y_3 \mapsto [Y_1] \\
  Z_2 \mapsto [\text{even}] \\
  \end{array}
  \] |

4.c.ii.C.II.2. Attempt to prove atomic predicate goal on top of goal stack.

4.c.ii.C.II.2.a. \( \forall E_{id} : \) Try assumption A5.
\( [Y_1] \neq [] \) \( [\text{even}] \neq [] \)
Unification failure: backtrack.

4.c.ii.C.II.2.b. \( \forall E_{id} : \) Try assumption A6, creating logic variables \( W_3, X_7, Y_4, Z_3 \).
\( W_3 \equiv \text{even} \) \( X_7 \equiv Y_1 \) \( Y_4 \equiv [] \) \( Z_3 \equiv [] \)
4.c.ii.C.II.2.b.i. \( \land I: \) Separate the conjunction into two separate goals.

### Goal Environment Equivalence Relation

<table>
<thead>
<tr>
<th>Goal</th>
<th>Environment</th>
<th>Equivalence Relation</th>
</tr>
</thead>
</table>
| \( \text{eq}(w, \text{even}) \land \text{iseven}(x) \land \text{parity}(y, z) \) | \[
    \begin{align*}
    w & \mapsto W_3 \\
    x & \mapsto X_7 \\
    y & \mapsto Y_4 \\
    z & \mapsto Z_3 \\
    \end{align*}
\] | \[
    \begin{align*}
    W_2 & \mapsto \text{odd} \\
    W_3 & \mapsto \text{even} \\
    X_1 & \mapsto X_4 \\
    X_4 & \mapsto s(X_6) \\
    X_5 & \mapsto \text{odd} \\
    X_6 & \mapsto z \\
    X_7 & \mapsto Y_1 \\
    Y_3 & \mapsto [Y_1] \\
    Y_4 & \mapsto [\] \\
    Z_2 & \mapsto [\text{even}] \\
    Z_3 & \mapsto [\] \\
\] |

### 4.c.ii.C.II.2.b.ii.

 Attempt to prove atomic predicate goal on top of goal stack.

4.c.ii.C.II.2.b.ii.A. \( \forall E_{\text{eq}}: \) Try assumption \( A_1 \), creating logic variable \( X_8 \).

\( X_8 \equiv \text{even} \)

Success! Continue with goal stack.

### Goal Environment Equivalence Relation

<table>
<thead>
<tr>
<th>Goal</th>
<th>Environment</th>
<th>Equivalence Relation</th>
</tr>
</thead>
</table>
| \( \text{iseven}(x) \land \text{parity}(y, z) \) | \[
    \begin{align*}
    w & \mapsto W_3 \\
    x & \mapsto X_7 \\
    y & \mapsto Y_4 \\
    z & \mapsto Z_3 \\
    \end{align*}
\] | \[
    \begin{align*}
    W_2 & \mapsto \text{odd} \\
    W_3 & \mapsto \text{even} \\
    X_1 & \mapsto X_4 \\
    X_4 & \mapsto s(X_6) \\
    X_5 & \mapsto \text{odd} \\
    X_6 & \mapsto z \\
    X_7 & \mapsto Y_1 \\
    Y_3 & \mapsto [Y_1] \\
    Y_4 & \mapsto [\] \\
    Z_2 & \mapsto [\text{even}] \\
    Z_3 & \mapsto [\] \\
\] |

4.c.ii.C.II.2.b.ii.B. \( \land I: \) Separate the conjunction into two separate goals.
<table>
<thead>
<tr>
<th>Goal</th>
<th>Environment</th>
<th>Equivalence Relation</th>
</tr>
</thead>
</table>
| iseven(x) | \[
\begin{align*}
    w & \mapsto W_3 \\
    x & \mapsto X_7 \\
    y & \mapsto Y_4 \\
    z & \mapsto Z_3
\end{align*}
\] | \[
\begin{align*}
    W_2 & \mapsto \text{odd} \\
    W_3 & \mapsto \text{even} \\
    X_1 & \mapsto X_4 \\
    X_4 & \mapsto s(X_6) \\
    X_5 & \mapsto \text{odd} \\
    X_6 & \mapsto z \\
    X_7 & \mapsto Y_1 \\
    X_8 & \mapsto \text{even} \\
    Y_3 & \mapsto [Y_1] \\
    Y_4 & \mapsto [] \\
    Z_2 & \mapsto [\text{even}] \\
    Z_3 & \mapsto []
\end{align*}
\] |
| parity(y,z) | \[
\begin{align*}
    w & \mapsto W_1 \\
    x & \mapsto X_7 \\
    y & \mapsto Y_4 \\
    z & \mapsto Z_3
\end{align*}
\] | \[
\begin{align*}
    W_2 & \mapsto \text{odd} \\
    W_3 & \mapsto \text{even} \\
    X_1 & \mapsto X_4 \\
    X_4 & \mapsto s(X_6) \\
    X_5 & \mapsto \text{odd} \\
    X_6 & \mapsto z \\
    X_7 & \mapsto Y_1 \\
    X_8 & \mapsto \text{even} \\
    Y_3 & \mapsto [Y_1] \\
    Y_4 & \mapsto [] \\
    Z_2 & \mapsto [\text{even}] \\
    Z_3 & \mapsto []
\end{align*}
\] |
| output | \[
\begin{align*}
    x & \mapsto X_1 \\
    y & \mapsto Y_1
\end{align*}
\] | \[
\begin{align*}
    X_1 & \mapsto X_4 \\
    X_4 & \mapsto s(X_6) \\
    X_5 & \mapsto \text{odd} \\
    X_6 & \mapsto z \\
    X_7 & \mapsto Y_1 \\
    X_8 & \mapsto \text{even} \\
    Y_3 & \mapsto [Y_1] \\
    Y_4 & \mapsto [] \\
    Z_2 & \mapsto [\text{even}] \\
    Z_3 & \mapsto []
\end{align*}
\] |

4.c.ii.C.II.2.b.ii.C.I. Attempt to prove atomic predicate goal on top of goal stack.

4.c.ii.C.II.2.b.ii.C.I.1. \( \forall E \text{Id} \): Try assumption A2.

\[Y_1 \equiv z\]

Success! Continue with goal stack.

4.c.ii.C.II.2.b.ii.C.II. Attempt to prove atomic predicate goal on top of goal stack.

4.c.ii.C.II.2.b.ii.C.II.1. \( \forall E \text{Id} \): Try assumption A5.

[[] \equiv []] [][[] \equiv []]

Success! Continue with goal stack.
4.c.ii.C.II.2.b.ii.C.II.2. Out
Output $x \mapsto s(z)$, $y \mapsto z$.
No more goals; solution found! Backtrack and find new solution.

This trace will continue infinitely, so we'll stop here. The interpreter would backtrack to 4.c.ii.C.II.2.b.ii.C and attempt to find another way to prove $\text{iseven}(x)$.