Small-Step Operational Semantics of `mini-prolog`

1 Introduction to `mini-prolog`

`mini-prolog` is a small, Prolog-like language which has been designed to require only a minimum of code in order to be implemented.

1.1 How it is Like Prolog

`mini-prolog` still bears the core features of Prolog, namely:

- Unification as a primitive of computation
- Nondeterministic execution, via a backtracking depth-first search strategy

1.2 How it is Unlike Prolog

For the sake of simplicity, certain features are intentionally missing:

- Negation as failure (`\+` or `not`). This requires reasoning about all successor states of a given state, which can severely complicate the semantics.
- The cut operator (`!`). Cut selectively retroactively removes choice points, and requires the semantics to track when choice points were added. This complicates the semantics.
- Predicates to dynamically alter the clause database, such as `assert` and `retract`. If the clause database is held constant, it allows for a simplification of the semantics.
- All meta-programming capabilities. Variables cannot be used as predicates, which simplifies both the syntax and the implementation. Second-order predicates like `forall` and `bagof` are not present, as these cannot be implemented in pure Prolog without dynamic modification of the clause database.

1.3 Basic Design

From the user's perspective, the syntax of `mini-prolog` looks just like Prolog, sans the key features previously mentioned. However, from the interpreter's perspective (which you will implement), `mini-prolog` looks a tad different from regular Prolog. Instead of operating directly on Prolog syntax (which we did for SimpleFUN), the interpreter operates on a more rigid *intermediate representation*. We arrive at this intermediate representation automatically via a translation process. With this in mind, the steps to go from input program to output are as follows:

1. Parse in user-facing Prolog syntax, yielding a Prolog AST
2. Translate the Prolog AST into an equivalent AST in our intermediate representation
3. Evaluate the intermediate representation

In practice, most language interpreters are written to operate over an intermediate representation as opposed to executing high-level syntax directly. For example, Java bytecode is viewable as an intermediate representation for the Java language, which is much more convenient to interpret than Java ASTs. In general, intermediate representations tend to make implicit operations explicit, breaking down complex operations into multiple simpler operations. Taking Java again as an example, Java bytecode distinguishes between allocating memory and invoking object constructors, which are both implicitly done at the same time when the `new` operator is used. This allows the Java interpreter to reason about object allocation separately from constructor invocation, which simplifies the implementation.

Specifically for `mini-prolog`, the biggest thing that the intermediate representation does is that it makes unification and nondeterminism very explicit, and it breaks them down into their most basic forms. With unification, `mini-prolog` has only a single operator that invokes unification, namely `=`. Moreover, `mini-prolog`'s `=` operator operates only over two variables, instead of between arbitrary Prolog terms. Both of these points are in contrast to regular Prolog, which allows unification to be performed implicitly between formal and actual parameters in calls, and also between arbitrary terms. With nondeterminism, `mini-prolog` only permits the choose (`;`) operator to add choice points. This is in contrast to regular Prolog, which additionally must handle implicit choices added when calls are performed to procedures consisting of multiple clauses.

2 `mini-prolog` Intermediate Representation Syntax

The intermediate representation of `mini-prolog` follows:

$$x \in \mathit{Variable} \qquad i \in \mathbb{Z} \qquad \mathit{name} \in \mathit{Name}$$

$$\begin{aligned}
p \in \mathit{Program} &::= \vec{c}\; q \\
q \in \mathit{Query} &::= \vec{x}\; b \\
c \in \mathit{Clause} &::= \mathit{name}(\vec{x}_1)\{\vec{x}_2\} \;{:}\!-\; b. \\
t \in \mathit{CompoundTerm} &::= \mathit{name}(\vec{x}) \\
b \in \mathit{Body} &::= b_1, b_2 \;\mid\; b_1 ; b_2 \;\mid\; x_1 = x_2 \;\mid\; \textsf{call}(t) \;\mid\; x \leftarrow t \;\mid\; x \;\textsf{is}\; e \;\mid\; x_1 \odot x_2 \;\mid\; \textsf{true} \;\mid\; \textsf{fail} \\
e \in \mathit{ArithExp} &::= i \;\mid\; x \;\mid\; e_1 \oplus e_2 \\
\oplus \in \mathit{ArithOp} &::= + \;\mid\; - \;\mid\; \times \;\mid\; \div \\
\odot \in \mathit{RelationalOp} &::= < \mid \leq \mid > \mid \geq \mid \texttt{=} \mid \neq
\end{aligned}$$

An English description of each of these syntactic components follows:

2.1 *Program*

A program (p) consists of a series of clauses, followed by a query. This is identical to Prolog.

2.2 *Query*

A query (q) consists of a series of variables, followed by a body. The series of variables lists the variables used in the body. It is guaranteed by the translation process that all variables in the list are used in the body, and that the body contains no variables which are not present in the list. To better understand this, consider the following Prolog query:

```
X = 1, Y = 2.
```

After the translation to the `mini-prolog` intermediate representation, the above query would look more like:

```
{X, Y} X = 1, Y = 2.
```

...where {`X, Y`} corresponds to the list of variables introduced in the body ($\vec{x}$ in $\vec{x}\; b$).

By forcing all variables which are used to be declared upfront, this simplifies how scoping works. With normal Prolog, we can introduce new variables at almost any point in the program. This is very inconvenient for an interpreter, because this means the environment (which holds which variables are in scope) can change at almost every step. By forcing variables to be introduced ahead of time, this means we only need to modify the environment at certain key points, simplifying the bulk of the implementation.

2.3 *Clause*

A clause (c) has a name, a number of formal parameters ($\vec{x}_1$), a number of local variables introduced in the body ($\{\vec{x}_2\}$), and the body itself (b). Compared two a normal Prolog clause, there are two key differences:

1. Any local variables used in the body (b) must be stated upfront (with $\{\vec{x}_2\}$)
2. The formal parameters may only be variables, as opposed to arbitrary Prolog terms. The translation process guarantees that each formal parameter is used only once in the list, even if the actual Prolog code used the same variable multiple times in the formal parameter list.

With the first difference, the motivation is identical to the motivation of forcing queries to state their variables upfront: the environment needs to change only at the start of a call, and not during subsequent execution of a call. With the second difference, the motivation is that this allows us to separate calls from unifications which occur due to calls. To better understand what this second difference gives us, consider the following program and query (denoted with `?-`):

```
foo(X, X).
?- foo(Y, 1).
```

When `foo` is called in the above query, it is expected that `Y` will be unified with `1` as part of the call. While this is semantically correct, it would mean that an interpreter would need to simultaneously handle the call to `foo` and perform the unification between `Y` and `1`. To simplify things, the translation rewrites the above code to something closer to the following:

```
foo(T1, T2) {X} :-
  T1 = X,
  T2 = X.
```

...where `X` is now treated as a pre-declared local variable, and the new variables `T1` and `T2` were introduced for the formal parameters. Thanks to this translation, actual parameters can always be blindly bound to formal parameters without full unification, as the translation guarantees that `T1` and `T2` will be new, previously unused variables. Moreoever, each formal parameter gets its own variable, so there is no need to perform any real unification at this stage. We instead delay unifications to the body of `foo`, which ultimately ensures that `T1` unifies with `T2`, which is done indirectly through the original formal parameter `X`.

2.4 *CompoundTerm*

Our compound terms look much like normal Prolog compound terms, except for the following key differences:

1. Each argument in a compound term must be a variable
2. The list of arguments is permitted to be empty

With the first difference, by forcing each parameter to be a variable, both the syntax and the implementation are greatly simplified. With the second difference, this effectively handles Prolog atoms (e.g., `foo` in `X = foo.`) without having a special case for atoms. That is, we treat atoms merely as compound terms with no arguments (i.e., zero-arity compound terms).

2.5 *Body*

For the most part, `mini-prolog` programs are built up from bodies. We divide the discussion of bodies up into a series of subsections, one for each possible kind of body.

2.5.1 And: b_1, b_2

The form b_1, b_2 states to first execute b_1, and if b_1 was true, then continue on to execute b_2. This works just as it does in Prolog.

2.5.2 Choose: $b_1; b_2$

The form $b_1; b_2$ states to nondeterministically execute b_1 and b_2, first exploring b_1 and then later exploring b_2 via a depth-first backtracking search. This behaves just as it does in Prolog. This is the **only** operation in `mini-prolog` which adds nondeterministic choice points.

2.5.3 Unify: $x_1 = x_2$

The form $x_1 = x_2$ states to unify the variables x_1 and x_2 with each other. This is the **only** operation in `mini-prolog` which causes unification to occur. Compared to regular Prolog, this unification applies strictly to two syntactic variables (more on that later).

2.5.4 Call: **call**(t)

The form **call**(t) states to call the procedure specified by the compound term t. For example, if we had the following Prolog code:

```
foo(X), bar(Y).
```

...this would get translated to something like the following:

```
call(foo(X)), call(bar(Y)).
```

As stated in the restrictions of `mini-prolog`, we can only call a known compound term t. This is in contrast to calling a compound term held in a variable, which would allow for dynamically constructing calls at runtime.

2.5.5 Bind: $x \leftarrow t$

The form $x \leftarrow t$ is effectively a highly specialized unification over two specific terms:

1. A variable x which has not yet appeared in the program
2. A compound term t

This form ultimately allows for variable to hold compound terms. To get an idea of how this operation works in practice, consider the following Prolog code:

```
foo(X, Y) = foo(bar, baz).
```

While the above code is valid Prolog, it is not valid `mini-prolog`, as `mini-prolog`'s unification can only be performed between two variables. However, with a combination of bind ($\leftarrow$) and `mini-prolog`'s unification operation, we can still achieve what Prolog does with the above snippet using only `mini-prolog`. This translation is shown below:

```
T1 ← foo(X, Y),
T2 ← bar(),
T3 ← baz(),
T4 ← foo(T2, T3),
T1 = T4.
```

There are several important points about the above translation:

- Both compound terms are incrementally built up, starting with inner components and working up to the whole compound term. The results of incremental steps are threaded through via temporary variables (i.e., variable `T1` through `T4`).
- The variable used on the lefthand side of each $\leftarrow$ operation has not been used in the program yet. This means that we do not need to do full unification with this lefthand variable, as we know ahead of time that the variable is currently uninstantiated.
- The Prolog atoms `bar` and `baz` are introduced as compound terms with zero arity.
- The actual unification between the compound terms is now performed indirectly through the variables `T1` and `T4`, which must be done as the final step (i.e., after the compound terms have actually been constructed).

2.5.6 Is: x **is** e

Much like Prolog, the **is** operation is used for evaluating an arithmetic expression e, binding the result to the variable x. As with the variable on the lefthand side of $\leftarrow$, the x in **is** is guaranteed to have never been seen before in the program. This means we do not need to perform full unification when putting the expression's value into variable x.

The **is** operation is also used for putting numbers into variables. For example, consider the following Prolog code:

```
X = 1.
```

This would get translated into the following `mini-prolog` code:

```
T1 is 1,
X = T1.
```

2.5.7 RelationalBody: $x_1 \odot x_2$

This operation is used to perform relational mathematical operations on two variables (e.g., `X > Y`). Unlike the closely related Prolog operator, this operator works **only** on variables. To illustrate how this translation works, consider the following Prolog snippet:

```
X > 7.
```

This would be translated as such:

```
T1 is 7,
X > T1.
```

2.5.8 True: **true**

true behaves just like it does in Prolog: it is a predicate which is trivially true. While **true** rarely shows up in regular Prolog code (given the fact that it does not do much of anything useful by itself), we will see it quite frequently during the execution of `mini-prolog` code. Ultimately, for each body element, we will evaluate down to either **true** or **fail**, and then respond accordingly.

2.5.9 Fail: **fail**

fail also behaves just as it does in Prolog: it is a trivially false predictae, which is used in practice to explicitly trigger backtracking to occur. **fail** will appear in execution whenever we determine that a more complex predicate is false.

2.6 *ArithExp*

This forms the different kinds of arithmetic expressions in `mini-prolog`. Specifically, we have the following kinds of expressions:

- Integers (i).
- Variables (x). If these do not evaluate to integers, then execution gets stuck.
- Binary arithmetic operations, $e_1 \oplus e_2$.

2.7 *ArithOp*

The different sorts of arithmetic operations allowed. We specifically allow addition ($+$), subtraction ($-$), multiplication ($\times$), and division ($\div$).

2.8 *RelationalOp*

The different sorts of operations that can be used in a relational body, which all operate over integers. The meaning of most of these operations should be straightforward. To be clear, the $=$ operator in this context does **not** perform unification; it simply checks that the two values are both integers which are equal to each other. Similarly, the $\neq$ operator simply checks that two integers hold different values, as opposed to behaving as something like a disequality operator (which we do not have in `mini-prolog`).

3 Translation from Prolog to `mini-prolog`

The internal syntax is arrived at via an automated translation process. This translation is responsble for the following tasks:

- Converting lists and list operations into structures and operations on structures. For example, the list `[1,2,3]` is represented as `.(1, .(2, .(3, [])))`, where "`.`" and "`[]`" are names. This naming is consistent with how actual Prolog engines treat lists, which are simply syntactic sugar over cons cells (`.`) and empty list (the atom named `[]`). This simplifies the semantics, which need not understand lists directly.

- Lifting any locally declared variables into a list of locally declared variables at the head of a query or the head of a clause. This allows the majority of the semantic rules to assume that all variables have already been declared, simplifying the rules. For example, if we have:

  ```
  foo(X) :-
    N = M,
    X = N.
  ```

  ...this would be translated to something like:

  ```
  foo(X) {N, M} :-
    N = M,
    X = N.
  ```

  ...where `{N, M}` is the list of variables which were locally introduced in `foo`. These variables are now effectively declared ahead of time, and used later. Note that this is not valid Prolog syntax.

- Reducing unification between any two values as a series of binds (i.e., ←), is (**is**) and unifications (i.e., `=`). For example, consider the following Prolog snippet:

  ```
  foo(X) :-
    X = foo(1).
  ```

  This would be translated into something like the following:

  ```
  foo(X) {T1, T2} :-
    T1 is 1,
    T2 ← foo(T1),
    X = T2.
  ```

  By permitting full unification only between variables, we can simplfy the syntax, and it means top-level unification only needs to consider the case where we compare two variables.

- Multiple clauses are combined together via the choose operator (`;`). For example, consider the following Prolog snippet:

  ```
  foo(1).
  foo(2).
  foo(3).
  ```

  This would be translated to something resembling the following:

  ```
  foo(X) :-
    (X = 1);
    (X = 2);
    (X = 3).
  ```

  This simplfies how nondeterminism is handled, because it means that the choose operator is the **only** operation in the language that adds choices. This also simplfies how calls are performed, as it means calls do not need to handle nondeterminism. (As you will see, calls are arguably the most complex operation in this language, so we do not want to make calls any harder than they already are.)

4 Small-Step Semantics

4.1 Notation

Specialized notation is used throughout the rest of this document. This subsection describes some of the more esoteric pieces.

4.1.1 $\mathcal{O}$ Meaning

$\mathcal{O}$ denotes an option type, written as the polymorphic type $\mathcal{O}(A) = \textsf{none} + \textsf{some}(A)$. This closely corresponds to Scala's `Option` class. Instances of $\mathcal{O}$ are represented by an o superscript. For example, i^o denotes a metavariable which holds either $\textsf{some}(i)$, where i is an integer, or $\textsf{none}$.

4.1.2 Lists and Sets

Consistent with previous handouts, the notation $\vec{v}$ denotes a list of values, and $\overrightarrow{Variable}$ denotes the *type* of a list of variables. The notation:

$$\overrightarrow{as} = a :: \overrightarrow{as}'$$

...indicates either list construction or list pattern matching, depending on the context. Specific to the above example, in a list construction context this prepends a to the list $\overrightarrow{as}'$, and binds it to the variable $\overrightarrow{as}$.

While the Scala code strictly differentiates between lists and sets, we treat the two somewhat interchangably here in certain contexts. For example, the notation $\vec{x}_1 \cup \vec{x}_2$ denotes either list concatenation or set union, depending on the particular context. For both sets and lists, the notation $a \in \overrightarrow{as}$ indicates a check to see if a exists in the set/list $\overrightarrow{as}$, and the related notation $a \notin \overrightarrow{as}$ is a check to see if a *does not exist* in the set/list $\overrightarrow{as}$.

4.1.3 Tuples

Tuples are either constructed or pattern matched against using the following notation:

$$(a \cdot b \cdot c)$$

...which denotes a 3-tuple holding the elements a, b, and c, in that order.

At the type level, tuples are represented with the following notation:

$$(A \times B \times C)$$

...which denotes a 3-tuple holding the types A, B, and C, in that order.

4.1.4 Maps

As with prior handouts, at the type level maps are represented as functions. For example, consider the following notation:

$$A \mapsto B$$

...which denotes a map which maps keys of type A to values of type B.

At the value level, maps have three significant built-in operations. These operations are as follows:

1. $\texttt{keys}(m)$: returns a set of the keys in map m.
2. $m(k)$: looks up the value which key k maps to in the map m. Gets stuck if $k \notin \texttt{keys}(m)$.
3. $m[k \mapsto v]$: returns a new map, which is the old map m updated with a mapping from the key k to the value v.

4.2 Semantic Domains (Data Structures)

The data structures used in these semantics are presented formally below. English descriptions of select components follow.

$$
\begin{aligned}
n &\in \mathbb{N} \\
db \in \mathit{ClauseDB} &= (\mathit{Name} \times \mathbb{N}) \mapsto \mathit{Clause} \\
\rho \in \mathit{Env} &= \mathit{Variable} \mapsto \mathit{Placeholder} \\
\kappa \in \mathit{Kont} &= \textsf{andK}(\mathit{Body}) + \textsf{restoreK}(\mathit{Env}) \\
v \in \mathit{Value} &= \mathit{IntValue} + \mathit{TermValue} + \mathit{Placeholder} \\
iv \in \mathit{IntValue} &= \textsf{intValue}(\mathbb{Z}) \\
tv \in \mathit{TermValue} &= \textsf{termValue}(\mathit{Name}, \overrightarrow{\mathit{Value}}) \\
pl \in \mathit{Placeholder} &= \textsf{placeholder}(\mathbb{N}) \\
eq \in \mathit{EquivClasses} &= \mathit{Placeholder} \mapsto \mathit{Value} \\
ch \in \mathit{Choice} &= \mathit{Body} \times \mathit{Env} \times \mathit{EquivClasses} \times \overrightarrow{\mathit{Kont}} \\
\varsigma \in \mathit{State} &= \mathit{Body} \times \mathit{Env} \times \mathit{EquivClasses} \times \overrightarrow{\mathit{Kont}} \times \overrightarrow{\mathit{Choice}}
\end{aligned}
$$

4.2.1 *ClauseDB*

The special domain *ClauseDB* is assumed to be globally available. This contains a database of clauses in the given program, organized as a mapping between clause names and arities to a clause with the given name and arity. To be clear, there is only ever one mapping of clause name/arity to clause in `mini-prolog`, as the translation took care of stitching multiple clauses with the same name/arity together with the choose (`;`) operator.

This database is created after parsing and translation, but before program execution. The database does not change throughout program execution. It is possible to refactor these semantics to pass around the database, but given the database's static nature such a refactoring would only needlessly complicate these semantics.

4.2.2 *Env*

Keeps track of which variables are in scope. In contrast to SimpleFUN, our variables do not map to typical values, but instead *Placeholder* values. *Placeholder* values are described in Section 4.2.5.

4.2.3 *Kont*

There are only two continuations:

1. **andK**(b): Representing the idea that we have to execute the body b later. This is used while executing b_1, b_2.
2. **restoreK**(ρ): Representing the idea that we later have to restore to the environment ρ later. This is used to restore the original variables in scope after a call is complete.

Collectively, these two continuations handle what is effectively the call stack.

4.2.4 *Value*

Values encode Prolog terms. For our purposes, there are only three kinds of values:

1. *IntValue*: Representation of a given integer.
2. *TermValue*: Representation of a compound term as a value. *TermValue*'s are associated with a given *Name*, and themselves contain a fixed number of *Value*'s. A *TermValue* containing no *Value* elements is equivalent to a Prolog atom (i.e., a symbol with a given name).
3. *Placeholder*: Effectively an internal variable representation, which is ultimately used to allow unification to be performed. These are described further in Section 4.2.5.

4.2.5 *Placeholder*

Placeholders are a special type of value which are unique to unification. Placeholders are used to implement equality constraints between two variables which still hold unknown values. To illustrate the problem placeholders solve, consider the following Prolog snippet:

```
X = Y,
X = 3.
```

It is expected that both `X` and `Y` hold the value `3` at the end of the above snippet, even though only `X` was explicitly assigned to `3`. This is because we first enforced that `X` and `Y` unify with each other (i.e., `X = Y`). With `X = Y`, we need to record that `X` and `Y` hold the same value, even though we do not know what this value is at this point in the program. This is done via careful manipulation of placeholder values. Intuitively, at the `X = Y`, we record that both variables hold the same placeholder value, which is a placeholder for some actual value we will assign later (specifically `3` in this case). The actual implementation is a bit more complex than this; we expand upon this further with the discussion of the *EquivClasses* data structure in Section 4.2.6.

4.2.6 *EquivClasses*

EquivClasses maintains equivalence classes between values, which is used for unification. This data structure ultimately tracks which variables have been unified with each other, allowing us to derive variable values. Placeholders are crucial to this process. To illustrate how this data structure works, consider again the Prolog program snippet:

```
X = Y,
X = 3.
```

At the start of the snippet, we will assign an unused placeholder value to each variable, and place this information in the environment. As such, initially the environment will contain:

$$\rho = [\texttt{X} \mapsto \textsf{\textbf{placeholder}}(0), \texttt{Y} \mapsto \textsf{\textbf{placeholder}}(1)]$$

...and the equivalence classes will be empty. Throughout execution, the environment will never change, though the equivalence classes will change significantly.

After executing `X = Y`, the equivalence classes will be updated to hold the following information:

$$eq = [\textsf{\textbf{placeholder}}(0) \mapsto \textsf{\textbf{placeholder}}(1)]$$

The interpretation of the above information is roughly as follows: "the value of **placeholder**(0) is the same as the value for **placeholder**(1)". In this case, **placeholder**(1) does not map to anything in eq, so we know that the value for **placeholder**(1) is still uninstantiated. However, we know that the value of **placeholder**(0) is the same as **placeholder**(1). Another way to say the same information is that **placeholder**(0) and **placeholder**(1) are in the same *equivalence class*, which is ultimately why the *EquivClasses* data structure has its particular name.

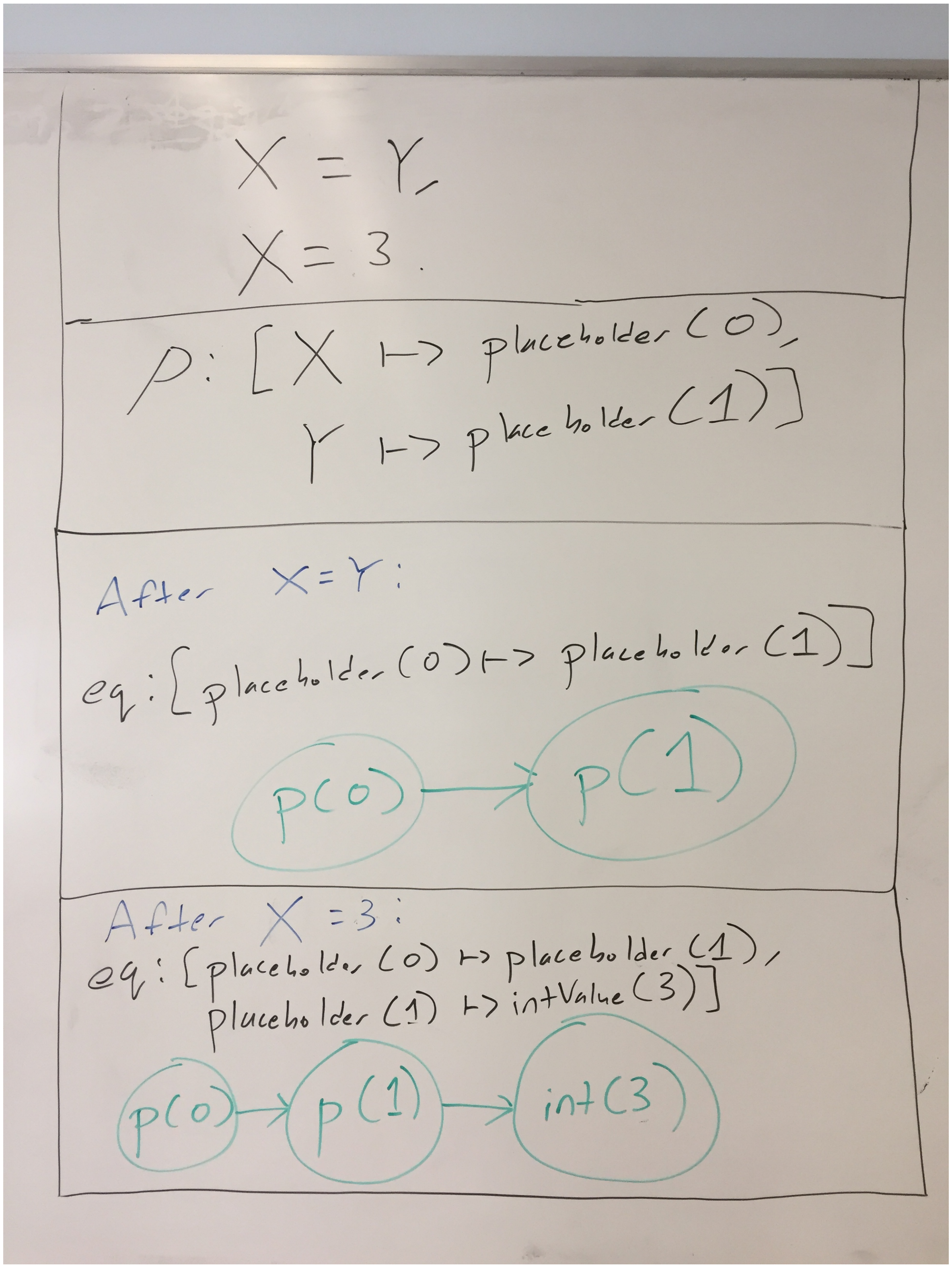
After executing the subsequent `X = 3` line, the equivalence classes will again be updated, this time to the following:

$$eq = [\textsf{\textbf{placeholder}}(0) \mapsto \textsf{\textbf{placeholder}}(1), \textsf{\textbf{placeholder}}(1) \mapsto \textsf{\textbf{intValue}}(3)]$$

The interpretation of the above information is roughly as follows: "the value of **placeholder**(0) is the same as the value of **placeholder**(1), and **placeholder**(1) has the integer value 3". That is, the values **placeholder**(0), **placeholder**(1), and **intValue**(3) are all in the same equivalence class. Looking a moment at the environment, this is ultimately how we know at program termination that both variable `X` and `Y` hold the same value of `3`. If we lookup `X` in the environment, we get **placeholder**(0). We then need to look up **placeholder**(0) in eq, yielding **placeholder**(1). The lookup process continues recursively until we reach either a non-placeholder or there is no mapping for the given placeholder; in this case, this means we look up **placeholder**(1) and see the mapping to **intValue**(3), at which point we know that variable `X`'s value is `3`. Similarly, for variable `Y`, we start this recursive lookup process in eq from **placeholder**(1), as `Y` maps to **placeholder**(1) in the environment.

The above information is shown on the next page in a graphical representation. The eq data structure is shown both via the typical mapping, as well as with a human-friendly graph representation in green.

We define the notation $eq[pl \equiv v]$ to indicate that pl should map to v in eq; that is, pl and v should be in the same equivalence class. This operation will return a new equivalence class holding a mapping between pl and v. This operation is defined by the `addRelation` helper function in Section 4.3.2. The notation $eq(v)$ will recursively lookup the given value in the equivalence class. This is defined by the `lookup` helper function in Section 4.3.1.


X = Y,
X = 3.
p: [X ↦ placeholder(0),
Y ↦ placeholder(1)]
After X = Y:
eq: [placeholder(0) ↦ placeholder(1)]
p(0)
p(1)
After X = 3:
eq: [placeholder(0) ↦ placeholder(1),
placeholder(1) ↦ intValue(3)]
p(0)
p(1)
int(3)

4.2.7 *Choice*

A *Choice* effectively freezes the state of the interpreter at some point in time. Choices are added while executing the $b_1; b_2$ operation, where b_1 is immediately executed, but b_2 is recorded as an alternative choice to be explored later. A *Choice* holds b_2, along with all the state needed to execute b_2, namely the environment (holding the variables in scope), the equivalence classes (holding how these variables map to values), and the continuation stack (holding what is effectively the call stack). Note that *Choice* elements do not themselves maintain the choice stack (i.e., $\overrightarrow{Choice}$), as this ends up being redundant if we are careful with how we push and pop choices from the choice stack.

4.2.8 *State*

As with SimpleFUN and small-step operational semantics in general, the state encodes everything the program needs to execute. The state consists of the following elements:

- *Body*: The body which is currently being executed (detailed in Section 2.5)
- *Env*: The variables which are in scope (detailed in Section 4.2.2)
- *EquivClasses*: How these variables map to values (detailed in Section 4.2.6
- $\overrightarrow{Kont}$: The continuation stack. As with SimpleFUN, the continuation stack effectively encodes what the call stack looks like. Continuations (*Kont*) are detailed in Section 4.2.3
- $\overrightarrow{Choice}$ The choice stack. Choices are pushed and popped, encoding a depth-first search over `mini-prolog` code. *Choice* elements are detailed in Section 4.2.7.

4.3 Helper Functions

4.3.1 `lookup`

Recursively looks up a value in *EquivClasses*.

$\texttt{lookup} \in \mathit{Value} \times \mathit{EquivClasses} \rightarrow \mathit{Value}$

$\texttt{lookup}(v, eq) =$

$$\begin{cases} \texttt{lookup}(eq(pl), eq) & \text{if } v = pl \wedge pl \in \texttt{keys}(eq) \\ v & \text{otherwise} \end{cases}$$

4.3.2 `addRelation`

Adds a relation to a given *EquivClasses* item. This is used to indicate that two values should be placed into the same equivalence class. This ends up being slightly more complex than adding a key/value pair to a map, as we must take some care to avoid adding cyclic relations.

$\texttt{addRelation} \in \mathit{Placeholder} \times \mathit{Value} \times \mathit{EquivClasses} \rightarrow \mathit{EquivClasses}$

$\texttt{addRelation}(pl, v, eq) =$

let $v' = \texttt{lookup}(v, eq)$

$$\begin{cases} eq & \text{if } pl = v' \\ eq[pl \mapsto v'] & \text{otherwise} \end{cases}$$

4.3.3 `unifyVals`

Unifies two *Value*s. This is at the heart of Prolog, and essentially allows for pattern matching to be performed between two data structures. This process can cause uninstantiated variables to become instantiated in both data structures. This can also cause aliasing to occur between uninstantiated variables. See `http://en.wikipedia.org/wiki/Unification_(computer_science)` for details about unification.

$\texttt{unifyVals} \in \mathit{Value} \times \mathit{Value} \times \mathit{EquivClasses} \rightarrow \mathcal{O}(\mathit{EquivClasses})$

$\texttt{unifyVals}(v_1, v_2, eq) =$

$\quad$ let $v_3 = \texttt{lookup}(v_1, eq)$

$\quad$ let $v_4 = \texttt{lookup}(v_2, eq)$

$$\begin{cases} \textsf{some}(\texttt{addRelation}(pl, v_4, eq)) & \text{if } v_3 = pl \\ \textsf{some}(\texttt{addRelation}(pl, v_3, eq)) & \text{if } v_3 \neq pl \wedge v_4 = pl \\ \textsf{some}(eq) & \text{if } v_3 = \textsf{intValue}(i) \wedge v_4 = \textsf{intValue}(i) \\ \texttt{unifyMultiVals}(\texttt{zip}(\vec{v_5}, \vec{v_6}), eq) & \text{if } v_3 = \textsf{termValue}(name, \vec{v_5}) \wedge v_4 = \textsf{termValue}(name, \vec{v_6}) \wedge |\vec{v_5}| = |\vec{v_6}| \\ \textsf{none} & \text{otherwise} \end{cases}$$

4.3.4 `unifyMultiVals`

For handling the unification of compound terms with the same name (*Name*) and arity.

$\texttt{unifyMultiVals} \in \overrightarrow{(\mathit{Value} \times \mathit{Value})} \times \mathit{EquivClasses} \rightarrow \mathcal{O}(\mathit{EquivClasses})$

$\texttt{unifyMultiVals}(\overrightarrow{values}, eq) =$

$$\begin{cases} \texttt{unifyMultiValsHelper}(\overrightarrow{values}', \texttt{unifyVals}(v_1, v_2, eq)) & \text{if } \overrightarrow{values} = (v_1 \cdot v_2) :: \overrightarrow{values}' \\ \textsf{some}(eq) & \text{otherwise} \end{cases}$$

4.3.5 `unifyMultiValsHelper`

Helper function used by `unifyMultiVals`.

$\texttt{unifyMultiValsHelper} \in \overrightarrow{(\mathit{Value} \times \mathit{Value})} \times \mathcal{O}(\mathit{EquivClasses}) \rightarrow \mathcal{O}(\mathit{EquivClasses})$

$\texttt{unifyMultiValsHelper}(\overrightarrow{values}, eq^o) =$

$$\begin{cases} \texttt{unifyMultiVals}(\overrightarrow{values}, eq) & \text{if } eq^o = \textsf{some}(eq) \\ \textsf{none} & \text{otherwise} \end{cases}$$

4.3.6 `callBodyEnvEq`

Helper function used for handling the **call** operation.

$\texttt{callBodyEnvEq} \in \overrightarrow{\mathit{Variable}} \times \mathit{Clause} \times \mathit{Env} \times \mathit{EquivClasses} \rightarrow (\mathit{Body} \times \mathit{Env} \times \mathit{EquivClasses})$

$\texttt{callBodyEnvEq}(\vec{x}_1, c, \rho_1, eq_1) =$

$\quad$ let $name(\vec{x}_2)\{\vec{x}_3\} :\!-\ b. = c$

$\quad$ let $\rho_2 = \texttt{newScope}(\vec{x}_2 \cup \vec{x}_3)$

$\quad$ let $eq_2 = \texttt{callBodyEnvEqHelper}(\vec{x}_2, \vec{x}_1, \rho_1, \rho_2, eq_1)$

$\quad$ $(b \cdot \rho_2 \cdot eq_2)$

4.3.7 `callBodyEnvEqHelper`

Helper function used by `callBodyEnvEq`.

$\texttt{callBodyEnvEqHelper} \in \overrightarrow{Variable} \times \overrightarrow{Variable} \times Env \times Env \times EquivClasses \rightarrow EquivClasses$

$\texttt{callBodyEnvEqHelper}(\vec{x}_1, \vec{x}_2, \rho_1, \rho_2, eq) =$

$$\begin{cases} \texttt{callBodyEnvEqHelper}(\vec{x}_3, \vec{x}_4, \rho_1, \rho_2, \texttt{addRelation}(\rho_2(x_5), \texttt{lookup}(\rho_1(x_6), eq), eq) & \text{if } \vec{x}_1 = x_5 :: \vec{x}_3 \wedge \vec{x}_2 = x_6 :: \vec{x}_4 \\ eq & \text{if } \vec{x}_1 = [] \wedge \vec{x}_2 = [] \end{cases}$$

4.3.8 `newScope`

Used to create a new scope, consisting of the given variables. Each variable will be mapped to a fresh placeholder value in the resulting environment.

$\texttt{newScope} \in \overrightarrow{Variable} \rightarrow Env$

$\texttt{newScope}(\vec{x}) =$

$\quad \texttt{toMap}(\texttt{map}(\vec{x}, x \Rightarrow (x \cdot \texttt{freshPlaceholder}())))$

4.3.9 `freshPlaceholder`

Gets a placeholder value which has not yet been used. From a high level, this means getting a placeholder value with a member of $\mathbb{Z}$ which has not yet been used. It is implemented in the Scala code with a mutable counter. The math can be adjusted to make this explicit by adding the counter value to the state, but this makes this a lot less clear.

$\texttt{freshPlaceholder} \in Unit \rightarrow Placeholder$

4.3.10 `toMap`

This is a mathematical representation of Scala's `toMap` method.

$\texttt{toMap} \in \overrightarrow{(A \times B)} \rightarrow (A \rightarrow B)$

4.3.11 `initialState`

Derives the initial program state, given a query to execute. In the math below, we use the notation {} to denote an empty *EquivClasses* data structure. In Scala, this is equivalent to the expression `EquivClasses(Map())`.

$\texttt{initialState} \in Query \rightarrow State$

$\texttt{initialState}(q) =$

$\quad \text{let } \vec{x}\ b = q$

$\quad (b \cdot \texttt{newScope}(\vec{x}) \cdot \{\} \cdot [] \cdot [])$

4.3.12 `evalCompoundTerm`

Evaluates a syntactic compound term down to a value.

$\texttt{evalCompoundTerm} \in CompoundTerm \times Env \times EquivClasses \rightarrow TermValue$

$\texttt{evalCompoundTerm}(t, \rho, eq) =$

$\quad \text{let } name(\vec{x}) = t$

$\quad \text{let } \vec{v} = \texttt{map}(\vec{x}, x \Rightarrow \texttt{lookup}(\rho x, eq))$

$\quad \textbf{termValue}(name, \vec{v})$

4.3.13 `evalArithExp`

Evaluates an arithmetic expression ($\mathit{ArithExp}$) down to an integer.

$\texttt{evalArithExp} \in \mathit{ArithExp} \times \mathit{Env} \times \mathit{EquivClasses} \rightarrow \mathbb{Z}$

$\texttt{evalArithExp}(e, \rho, eq) =$

$$\begin{cases} i & \text{if } e = i \\ i & \text{if } e = x \wedge \texttt{lookup}(\rho(x), eq) = \textsf{\textbf{intValue}}(i) \\ \texttt{evalArithExpHelper}(\texttt{evalArithExp}(e_1, \rho, eq), \oplus, \texttt{evalArithExp}(e_2, \rho, eq)) & \text{if } e = e_1 \oplus e_2 \end{cases}$$

4.3.14 `evalArithExpHelper`

Helper function used by `evalArithExp`.

$\texttt{evalArithExpHelper} \in \mathbb{Z} \times \oplus \times \mathbb{Z} \rightarrow \mathbb{Z}$

$\texttt{evalArithExpHelper}(i_1, \oplus, i_2) =$

$$\begin{cases} i_1 \mathbin{\hat{+}} i_2 & \text{if } \oplus = \texttt{+} \\ i_1 \mathbin{\hat{-}} i_2 & \text{if } \oplus = \texttt{-} \\ i_1 \mathbin{\hat{\times}} i_2 & \text{if } \oplus = \times \\ i_1 \mathbin{\hat{\div}} i_2 & \text{if } \oplus = \div \wedge i_2 \neq 0 \end{cases}$$

4.3.15 `relationalHolds`

Effectively evaluates relational bodies, given the integers being compared along with the actual relational operation to use (e.g., $<$).

$\texttt{relationalHolds} \in \mathbb{Z} \times \mathit{RelationalOp} \times \mathbb{Z} \rightarrow \mathit{Boolean}$

$\texttt{relationalHolds}(i_1, \odot, i_2) =$

$$\begin{cases} i_1 \mathrel{\hat{<}} i_2 & \text{if } \odot = \texttt{<} \\ i_1 \mathrel{\hat{\leq}} i_2 & \text{if } \odot = \leq \\ i_1 \mathrel{\hat{>}} i_2 & \text{if } \odot = \texttt{>} \\ i_1 \mathrel{\hat{\geq}} i_2 & \text{if } \odot = \geq \\ i_1 \mathrel{\hat{=}} i_2 & \text{if } \odot = \texttt{=} \\ i_1 \mathrel{\hat{\neq}} i_2 & \text{if } \odot = \neq \end{cases}$$

4.3.16 `map`

Applies the given function to each element of the provided list, as per the usual definition of `map`. This is considered basic, and so has not been formalized.

$\texttt{map} \in \vec{A} \times (A \rightarrow B) \rightarrow \vec{B}$

4.4 Inference Rules

For each rule below, the following variables are always in scope:

- b: The body in the current state
- ρ: The environment in the current state
- eq: The equivalence classes of the current state
- $\vec{\kappa}$: The continuation stack for the current state
- $\overrightarrow{ch}$: The choice stack for the current state
- db: The clause database, which is global (i.e., not specific to a single state)

As usual, each rule specifies the new values of each of these corresponding values in the next state.

#	b	Premises	b_{new}	ρ_{new}	eq_{new}	$\overrightarrow{\kappa_{new}}$	$\overrightarrow{ch_{new}}$
1	b_1, b_2		b_1	ρ	eq	$\textsf{andK}(b_2) :: \vec{\kappa}$	$\overrightarrow{ch}$
2	$b_1; b_2$		b_1	ρ	eq	$\vec{\kappa}$	$(b_2 \cdot \rho \cdot eq \cdot \vec{\kappa}) :: \overrightarrow{ch}$
3	$x_1 = x_2$	$\texttt{unifyVals}(\rho(x_1), \rho(x_2), eq) = \textsf{some}(eq')$	$\textsf{true}$	ρ	eq'	$\vec{\kappa}$	$\overrightarrow{ch}$
4	$x_1 = x_2$	$\texttt{unifyVals}(\rho(x_1), \rho(x_2), eq) = \textsf{none}$	$\textsf{fail}$	ρ	eq	$\vec{\kappa}$	$\overrightarrow{ch}$
5	$\textsf{call}(t)$	$t = name(\vec{x}), c = db((name \cdot \lvert\vec{x}\rvert))$, $(b' \cdot \rho' \cdot eq') = \texttt{callBodyEnvEq}(\vec{x}, c, \rho, eq)$	b'	ρ'	eq'	$\textsf{restoreK}(\rho) :: \vec{\kappa}$	$\overrightarrow{ch}$
6	$x \leftarrow t$	$tv = \texttt{evalCompoundTerm}(t, \rho, eq)$, $pl = \texttt{lookup}(\rho(x), eq)$, $eq' = \texttt{addRelation}(pl, tv, eq)$	$\textsf{true}$	ρ	eq'	$\vec{\kappa}$	$\overrightarrow{ch}$
7	x **is** e	$i = \texttt{evalArithExp}(e, \rho, eq)$, $v = \textsf{intValue}(i)$, $pl = \texttt{lookup}(\rho(x), eq)$, $eq' = \texttt{addRelation}(pl, v, eq)$	$\textsf{true}$	ρ	eq'	$\vec{\kappa}$	$\overrightarrow{ch}$
8	$x_1 \odot x_2$	$\textsf{intValue}(i_1) = \texttt{lookup}(\rho(x_1), eq)$, $\textsf{intValue}(i_2) = \texttt{lookup}(\rho(x_2), eq)$, $\texttt{relationalHolds}(i_1, \odot, i_2)$	$\textsf{true}$	ρ	eq	$\vec{\kappa}$	$\overrightarrow{ch}$
9	$x_1 \odot x_2$	$\textsf{intValue}(i_1) = \texttt{lookup}(\rho(x_1), eq)$, $\textsf{intValue}(i_2) = \texttt{lookup}(\rho(x_2), eq)$, $\neg\texttt{relationalHolds}(i_1, \odot, i_2)$	$\textsf{fail}$	ρ	eq	$\vec{\kappa}$	$\overrightarrow{ch}$
10	$\textsf{true}$	$\vec{\kappa} = \textsf{andK}(b') :: \vec{\kappa}'$	b'	ρ	eq	$\vec{\kappa}'$	$\overrightarrow{ch}$
11	$\textsf{true}$	$\vec{\kappa} = \textsf{restoreK}(\rho') :: \vec{\kappa}'$	$\textsf{true}$	ρ'	eq	$\vec{\kappa}'$	$\overrightarrow{ch}$
12	$\textsf{true}$	$\vec{\kappa} = [], \overrightarrow{ch} = (b' \cdot \rho' \cdot eq' \cdot \vec{\kappa}') :: \overrightarrow{ch}'$	b'	ρ'	eq'	$\vec{\kappa}'$	$\overrightarrow{ch}'$
13	$\textsf{true}$	$\vec{\kappa} = [], \overrightarrow{ch} = []$	b	ρ	eq	$\vec{\kappa}$	$\overrightarrow{ch}$
14	$\textsf{fail}$	$\overrightarrow{ch} = (b' \cdot \rho' \cdot eq' \cdot \vec{\kappa}') :: \overrightarrow{ch}'$	b'	ρ'	eq'	$\vec{\kappa}'$	$\overrightarrow{ch}'$
15	$\textsf{fail}$	$\overrightarrow{ch} = []$	b	ρ	eq	$\vec{\kappa}$	$\overrightarrow{ch}$