Interpreters and Metainterpreters

1 Summary

This handout introduces the concept of a Prolog *metainterpreter*; that is, an interpreter written in Prolog. First, some necessary background is provided on interpreters and general, followed by a discussion of Prolog metainterpreters.

2 Background: Program Interpreters

An *interpreter* is a program that takes in another program as input, and executes the given program. This is a general concept which is widely applicable. While this might sound abstract, you've both seen and written multiple interpreters at this point. Specifically, the code for evaluating arithmetic and boolean expressions would qualify as a type of interpreter, and the arithmetic/boolean expressions they take as input would qualify as (albeit simplistic) programs.

> ***Sidenote:*** *Compilers* work differently than interpreters, though both tend to get used in the same capacity. Like interpreters, compilers are programs that take programs as input. However, the output of a compiler is another program, usually written in a different language than the input program. For example, C compilers generally produce assembly code as output, given C code as input. This is very different from the output of an interpreter, as the output of an interpreter is *the output of the input program itself.* Viewed another way, interpreters execute input programs, whereas compilers translate input programs into other programs.

We can build more advanced interpreters which evaluate more rich languages using the same sort of concepts you've already seen with arithmetic and boolean evaluators. For example, consider the following program grammar:

$$b \in \textit{Boolean} ::= \textbf{true} \mid \textbf{false} \qquad i \in \mathbb{Z}$$
$$e \in \textit{Expression} ::= i \mid b \mid e_1 < e_2 \mid e_1 \;\&\&\; e_2 \mid \textbf{if}\ (e_1)\ \{e_2\}\ \textbf{else}\ \{e_3\} \mid \textbf{print}(e)$$

While the above language is still relatively simplistic, this is arguably much more expressive than before. Notably, the above language combines both arithmetic and boolean operations, adds in an **if** expression, and even allows for the program to output values with **print**. We can now ask questions like "is 5 less than 6?" $(5 < 6)$, and perform different operations depending on the result. Overall, the above language can be described by the following informal semantics:

- Integers (i) evaluate to themselves
- Booleans (b) evaluate to themselves
- $e_1 < e_2$ evaluates to **true** if e_1 evaluates to an integer which is less than the integer produced by evaluating e_2 (e.g., $5 < 6$). If the integer that e_1 evaluates to is $\geq$ the integer e_2 evaluates to, then $e_1 < e_2$ evaluates to **false**. If either e_1 or e_2 does not evaluate to an integer (e.g., $5 <$ **true**), then evaluation stops and the program crashes.
- e_1 && e_2 evaluates to **true** if both e_1 and e_2 evaluate to the boolean **true** (e.g., **true** && $(1 < 2)$). If both e_1 and e_2 evaluate to any different combination of booleans, then e_1 && e_2 evaluates to **false** (e.g., $(3 < 3)$ && $(1 < 2)$). If either e_1 or e_2 does not evaluate to a boolean (e.g., 5 && **true**), then evaluation stops and the program crashes.
- **if** (e_1) $\{e_2\}$ **else** $\{e_3\}$ evaluates down to whatever e_2 evaluates down to if e_1 evaluates down to the boolean **true**. If e_1 evaluates down to the boolean **false**, then **if** (e_1) $\{e_2\}$ **else** $\{e_3\}$ evaluates down to whatever e_3 evaluates down to. If e_1 evaluates down to something that isn't a boolean, then evaluation stops and the program crashes.
- **print**(e) will print out whatever e evaluates down to. **print**(e) itself will evaluate down to the special value **void**.

We can implement an interpreter in Prolog for the above language. However, before we implement the interpreter, let's redefine the grammar to be a little more amenable to definition in Prolog. Specifically, we'll redefine everything to work in terms of structures and atoms. Such a redefinition is below:

$$i \in \mathbb{Z}$$

$$b \in \textit{Boolean} ::= \textbf{true} \mid \textbf{false}$$

$$e \in \textit{Expression} ::= \textbf{integer}(i) \mid \textbf{boolean}(b) \mid \textbf{lessThan}(e_1, e_2) \mid \textbf{and}(e_1, e_2) \mid \textbf{if}(e_1, e_2, e_3) \mid \textbf{print}(e)$$

Now that everything in this grammar is defined in terms of atoms and structures, this AST can be more easily represented in Prolog.

> ***Sidenote:*** Note that this change to the above grammar is relatively inconsequential. Most importantly, the semantics of the language will not change. Additionally, such a conversion is usually the job of a *parser*, which reads an input file (viewable as a single giant `String` value) and converts it into an abstract syntax tree (AST). The specific AST representation depends on whatever language the interpreter is being written in; in this case, since we're using Prolog, we're using a grammar/AST definition which is amenable to Prolog. We *could* implement a parser in Prolog as well, though this is tangential to the problem at hand.

We incrementally define our Prolog-based interpreter below. First, we define two Prolog facts corresponding to how integers and booleans should be evaluated:

```
1   eval(integer(N), int(N)).
2   eval(boolean(B), bool(B)).
```

These rules state that integers (line 1) and booleans (line 2) evaluate to themselves, respectively. In this case, we use structures named **int** and **bool** to represent the evaluated values. This distinguishes integer ASTs from integer values, and so on. Such a distinction isn't absolutely necessary, but it can help with debugging (without this, it can be difficult to distinguish between ASTs and values, which are distinct things).

The next thing implemented is a rule handling **lessThan**, shown below:

```
1 eval(lessThan(E1, E2), bool(Result)) :-
2     eval(E1, int(E1Result)),
3     eval(E2, int(E2Result)),
4     (E1Result < E2Result ->
5         Result = true;
6         Result = false).
```

A line-by-line description of this rule follows:

1. `lessThan(E1, E2)` evaluates to a boolean
2. `E1` evaluates to an integer
3. `E2` evaluates to an integer
4. If the first integer is less than the second integer...
5. ...if so, then the result is `true`
6. ...if not, then the result is `false`

For the next rule (`and`), we'll first introduce a helper with the same name, shown below:

```
1 and(false, _, false).
2 and(_, false, false).
3 and(true, true, true).
```

This is the same code implementing boolean AND from the boolean evaluator; this ultimately states that only `true` AND `true` yield `true`, whereas everything else yields `false`.

With the above `and` helper in hand, we'll implement the rule handling `and` AST nodes, shown below:

```
1  eval(and(E1, E2), bool(Result)) :-
2      eval(E1, bool(E1Result)),
3      eval(E2, bool(E2Result)),
4      and(E1Result, E2Result, Result).
```

A line-by-line explanation of the above code follows:

1. `and(E1, E2)` evaluates to a boolean value
2. `E1` must evaluate down to a boolean value
3. `E2` must evaluate down to a boolean value
4. The final result is determined by performing boolean AND on `E1Result` (derived by evaluating `E1`) and `E2Result` (derived by evaluating `E2`)

From here, we'll implement the rule for `if`, shown below:

```
1  eval(if(E1, E2, E3), Result) :-
2      eval(E1, bool(B)),
3      (B == true ->
4          eval(E2, Result);
5          eval(E3, Result)).
```

A line-by-line description follows:

1. `if(E1, E2, E3)` evaluates down to some result
2. `E1` evaluates down to a boolean `B`
3. If `B`...
4. ...is `true`, then the final result is whatever `E2` evaluates down to
5. ...is `false` (well, technically any value except `true`, but booleans can only be `true` or `false`), then the final result is whatever `E3` evaluates down to

We finally handle the case for `print`, shown below:

```
1  eval(print(E), void) :-
2      eval(E, ToPrint),
3      writeln(ToPrint).
```

A line-by-line description follows:

1. `print(E)` evaluates to the special value `void`
2. Evaluate `E` down to some arbitrary result `ToPrint`
3. Write out `ToPrint`, followed by a newline

The interpreter code in its entirety is shown below:

```
1  and(false, _, false).
2  and(_, false, false).
3  and(true, true, true).
4
5  eval(integer(N), int(N)).
6  eval(boolean(B), bool(B)).
7  eval(lessThan(E1, E2), bool(Result)) :-
8      eval(E1, int(E1Result)),
9      eval(E2, int(E2Result)),
10     (E1Result < E2Result ->
11         Result = true;
12         Result = false).
13 eval(and(E1, E2), bool(Result)) :-
14     eval(E1, bool(E1Result)),
15     eval(E2, bool(E2Result)),
16     and(E1Result, E2Result, Result).
17 eval(if(E1, E2, E3), Result) :-
18     eval(E1, bool(B)),
19     (B == true ->
20         eval(E2, Result);
21         eval(E3, Result)).
22 eval(print(E), void) :-
23     eval(E, ToPrint),
24     writeln(ToPrint).
```

We can issue queries to the above code, which amounts to giving this interpreter programs to interpret. For example, consider the following program, written in the style of the first (non-Prolog-friendly) grammar:

```
if ((5 < 6) && true) {
  print(1)
} else {
  print(2)
}
```

The above AST is equivalently represented in the Prolog-friendly grammar as such:

```
if(
  and(lessThan(integer(5), integer(6)),
      boolean(true)),
  print(integer(1)),
  print(integer(2)))
```

We can interpret the above AST as a program, as shown below:

```
1 ?- eval(
2    if(
3      and(lessThan(integer(5), integer(6)),
4          boolean(true)),
5      print(integer(1)),
6      print(integer(2))),
7    Result).
8 int(1)
9 Result = void.
```

Line 8 above shows that this program output `int(1)`, indicating that the first `print` (specifically the one on line 5) was executed. This is expected considering that $(5 < 6)$ && **true** evaluates to **true**, ultimately causing `print(1)` to be executed. Line 6 shows that the AST evaluates down to **void**, the special value that `print` expressions always evaluate to.

3 A Basic Prolog Metainterpreter

We can take this same sort of approach to defining an interpreter for Prolog. However, instead of doing everything from scratch (as previously done), we'll instead take a major shortcut and define the Prolog interpreter *in* Prolog. This will allow us to piggyback off of Prolog's features as needed, dramatically cutting down on the amount of code we need to write.

First, let's start with a grammar describing a subset of Prolog we are interested, which itself will be written in a Prolog-friendly form:

$$c \in \mathit{Clause} ::= \textbf{true} \mid t_1 \textbf{ is } t_2 \mid \overset{t \in \mathit{Term}}{t_1 = t_2} \mid c_1, c_2 \mid c_1; c_2 \mid \backslash\!+ c \mid t$$

We leave the definition of *terms* (effectively Prolog data: variables, atoms, and structures) abstract, as we will piggyback off of Prolog for the full definition. From here, we define a number of clauses that we are interested in handling, namely:

- **true**, which is a clause that trivially succeeds without doing any real work.
- t_1 **is** t_2, representing **is** with its usual meaning of arithmetic evaluation. Note that this is equivalently written as a structure: **is**(t_1, t_2).
- $t_1 = t_2$, with its usual meaning of unification. This is equivalently written as the structure $=(t_1, t_2)$.
- c_1, c_2, with its usual meaning of conjunction. This is equivalently written as the structure $,(c_1, c_2)$.
- c_1; c_2, with its usual meaning of disjunction (that is, splitting the world). This is equivalently written as the structure $;(c_1, c_2)$.
- $\backslash\!+ c$, with its usual meaning of negation-as-failure. This is equivalently written as the structure $\backslash\!+(c)$.
- Terms on their own, t, which represent calls. For example, `append(L1, L2, L3)` is a call if used in the context of a clause. In fact, it is this context which defines whether or not a given Prolog term is code or data; everything shown can be either code or data, depending on how it is used (`append` can just as easily be data, as in `X = append(L1, L2, L3)`).

We incrementally define a fully-functioning interpreter for the above language, using Prolog itself. First, the case for **true**:

```
interpret(true) :- !.
```

As shown, **true** merely succeeds, but with a slight twist: there is a cut (`!`). Every rule we implement is going to begin with this cut, which serves to prevent the interpreter itself from backtracking over different productions. At the moment, the need for this cut may not seem apparent; if you do not understand its purpose, don't worry quite yet.

For the next rule, we will handle **is**, shown below:

```
1 interpret((A is B)) :-
2     !,
3     A is B.
```

On line 1, we show that this rule applies if the input is an **is**. The extra parentheses are needed to help Prolog correctly parse this. This first line could equivalently be written as a usual structure, like so: `interpret(is(A, B))`. The second line performs the same sort of cut as before, with the same purpose. The third line defers how **is** is handled to the built-in **is** operation. This third line is where we first exploit the fact that this is a *meta*interpreter: because **is** is already available, we can just use the built-in definition of **is** without defining our own. If we needed to define our own, we'd have to define exactly how things like addition and multiplication work, which would take quite a bit of code.

Unification is defined quite similarly to **is**, shown below:

```
1 interpret((A = B)) :-
2     !,
3     A = B.
```

As before, we defer to Prolog's built-in definition of unification for this task. This saves arguably even more code than we saved with **is**, as full unification takes a fair amount of work to implement.

Next we will implement conjunction, shown below:

```
1 interpret((A, B)) :-
2     !,
3     interpret(A),
4     interpret(B).
```

This also makes great use of Prolog's built-in definition of conjunction. As written, this code recursively calls `interpret` to interpret clause `A` (line 3), and then uses built-in conjunction (`,`) to interpret clause `B` with another recursive call to `interpret` (line 4). The most English-like translation of the above code is the statement "To interpret A, B, we need to interpret A, and then interpret B.". Because we are defining this interpreter directly in Prolog, we quite conveniently have access to something implementing the "and then" part above (namely conjunction: `,`).

Disjunction and negation-as-failure are implemented in the exact same way as with conjucntion, except they defer the appropriate built-in Prolog operations. These are both shown below:

```
1 interpret((A; B)) :-
2     !,
3     (interpret(A); interpret(B)).
4 interpret(\+(A)) :-
5     !,
6     \+ interpret(A).
```

The meaning of the above code should be clear, as this follows an identical pattern to conjunction (`,`).

The most complex case to handle is that of calls. For calls, we will make use of a special built-in routine: `clause/2` (`http://www.swi-prolog.org/pldoc/man?predicate=clause/2`). The `clause/2` procedure takes a term representing a call, and will nondeterministically perform the first step of a call, putting what's left to execute in the second . This is best explained via an example. Consider the following rulebase:

```
1 foo(1).
2 foo(2) :-
3     8 is 4 + 4.
4 foo(X) :-
5     27 = 27,
6     NewX is X - 1,
7     foo(NewX).
8
9 bar(17).
```

With the above rulebase in hand, let's perform some queries involving `clause/2`. First, a query involving `foo(1)`:

```
1   ?- clause(foo(1), Body).
2   Body = true ;
3   Body = (27=27, _G2359 is 1+ -1, foo(_G2359)).
```

As shown, there were two answers produced from this query, described in order:

1. The call `foo(1)` unifies with the fact `foo(1)` in the rulebase. The `Body`, that is, what is remaining to be executed, is simply `true`, which always just evaluates to true without doing any special execution. This use of `true` here, incidentally, is why we implement `true` in our metainterpreter in the first place.

2. The call `foo(1)` also unifies with `foo(X)`, leading to a second answer. In this case, the body is much more complex: we have a conjunction of things left to execute. The variable `X` has been filled in with `1` in this body, as we know from the call `foo(1)` that `X` must be `1`. The variable `NewX` is present in this body, though the name has been mangled to be something else (namely `_G2359`). However, `NewX` still holds the same meaning; this can be determined by the fact that `_G2359` is used twice, meaning these are both the same variable. While `NewX` (`_G2359`) will eventually get a value, this will only occur once `NewX is X - 1` (`_G2359 is 1+ -1`) is executed, and this clause hasn't yet been executed.

Missing from the possible answers is code pertaining to `foo(2)` and `bar(17)`. This is because neither of these terms unify with the term `foo(1)`. Thinking in terms of how calls work, `foo(2)` is a procedure that doesn't have a compatible *head* as our call `foo(1)`, and `bar` is an entirely separate procedure.

With `clause/2` in hand, the implementation of calls becomes relatively short:

```
1   interpret(Call) :-
2     clause(Call, Body),
3     interpret(Body).
```

As shown on line 1, a call can be any term, so we simply use the variable `Call` here. This is why the cut (`!`) was needed in the other rules: without the cut, Prolog could backtrack with something that definitely isn't a call (e.g., conjunction) and try to treat it as a call. This would lead to our metainterpreter crashing, as `clause` only works with things that are calls. At line 2 above, `clause/2` is used to make the first step of a call. This will nondeterministically select between all the clauses that could be called here, binding the remainder of what needs to be executed to the variable `Body`. Conveniently, `clause/2`'s nondeterministic solutions are always in order of clauses present in the file, so the above code will end up exploring solutions in the exact same order as Prolog will. Line 3 will interpret what's left to execute (stored in `Body`) using a recursive call to `interpret`.

The complete metainterpreter is shown below, along with a definition of `myLength` which will recursively compute the length of an input list. This definition will be used with some example queries in a moment.

```
1  interpret(true) :- !.
2  interpret((A is B)) :-
3      !,
4      A is B.
5  interpret((A = B)) :-
6      !,
7      A = B.
8  interpret((A, B)) :-
9      !,
10     interpret(A),
11     interpret(B).
12 interpret((A; B)) :-
13     !,
14     (interpret(A); interpret(B)).
15 interpret(\+(A)) :-
16     !,
17     \+ interpret(A).
18 interpret(Call) :-
19     clause(Call, Body),
20     interpret(Body).
21
22 myLength([], 0).
23 myLength([_|T], Len) :-
24     myLength(T, TLen),
25     Len is TLen + 1.
```

Example queries follow. First, a basic check that unification is working:

```
?- interpret((A = 5)).
A = 5.
```

As shown, we call the `interpret` procedure with the data `A = 5`, which is equivalent to calling `interpret` with the structure `=(A, 5)`. Unification is performed between `A` and `5`, leading to the expected solution that `A = 5`.

This sort of unification can be chained along with conjunction, like so:

```
?- interpret((A = 5, B is 4 + A)).
A = 5,
B = 9.
```

As shown, both `A = 5` and `B is 4 + A` are executed, leading to the solution that `A = 5` and `B = 9`.

This works for calls as well, as shown below:

```
?- interpret(myLength([foo, bar, baz], Len)).
Len = 3.
```

That is, the above code calls `myLength` with parameters `[foo, bar, baz]` and `Len`, leading to the expected answer that `Len = 3` (the length of the input list).

4 Exploiting Metainterpreters: Bounding

As shown in the previous section, Prolog makes it relatively easy to define a fully-functional metainterpreter. Such a definition for a basic interpreter is mostly an academic exercise, but we can exploit this ease to effectively redefine the basics of how Prolog works. For example, let's consider again the problem of defining a test case generator, as shown below:

```
1  exp(integer(0)).
2  exp(plus(E1, E2)) :-
3    exp(E1),
4    exp(E2).
```

As written, the above procedure will keep "spamming" on the rightmost AST node, leading us to very skewed ASTS:

```
1  ?- exp(E).
2  E = integer(0) ;
3  E = plus(integer(0), integer(0)) ;
4  E = plus(integer(0), plus(integer(0), integer(0))) ;
5  E = plus(integer(0), plus(integer(0), plus(integer(0), integer(0))))exp(integer(0)) ;
6  ...
```

We have been fixing this problem up until now by adding a `decBound` procedure into the mix, and decreasing some arbitrary bound each time a recursive call is made. However, instead of doing such bounding in the `exp` procedure itself, we can instead *redefine Prolog* to do this bounding automatically for us. This is done with a custom metainterpreter that will call `decBound` when the *interpreter itself* makes a recursive call. Such an interpreter is shown below:

```
1  decBound(In, Out) :-
2      In > 0,
3      Out is In - 1.
4
5  interpret(_, true) :- !.
6  interpret(_, (A is B)) :-
7      !,
8      A is B.
9  interpret(_, (A = B)) :-
10     !,
11     A = B.
12 interpret(Bound, (A, B)) :-
13     !,
14     decBound(Bound, NewBound),
15     interpret(NewBound, A),
16     interpret(NewBound, B).
17 interpret(Bound, (A; B)) :-
18     !,
19     decBound(Bound, NewBound),
20     (interpret(NewBound, A); interpret(NewBound, B)).
21 interpret(Bound, \+(A)) :-
22     !,
23     decBound(Bound, NewBound),
24     \+ interpret(NewBound, A).
25 interpret(Bound, Call) :-
26     decBound(Bound, NewBound),
27     clause(Call, Body),
28     interpret(NewBound, Body).
```

With this custom metainterpreter in hand, we can now bound automatically without redefining `exp`, like so:

```
?- interpret(5, exp(E)).
E = integer(0) ;
E = plus(integer(0), integer(0)) ;
E = plus(integer(0), plus(integer(0), integer(0))) ;
E = plus(plus(integer(0), integer(0)), integer(0)) ;
E = plus(plus(integer(0), integer(0)), plus(integer(0), integer(0))) ;
false.
```

As shown above, `false` is produced, indicating that this truly does exhaust the entire space.

You might have noticed that the above answers are a little off with respect to the bound. A bound of `5` was previously generation expressions up to depth 5, but none of the above expressions are quite that deep. This is because what we're bounding here is instead of the recursion bound of the *interpreter itself*, as opposed to the recursion bound of `exp`. This means that things like conjunction are bounded, even though they are innocuous. This is easily amended by defining a different kind of custom metainterpreter, one that bounds specifically at calls and keeps track of the call stack (specifically the things that have been called before). However, this is left as an exercise to the reader. As a hint, if you're interested in implementing something like this, you may want to take a look at the `functor/3` built-in procedure (`http://www.swi-prolog.org/pldoc/doc_for?object=functor/3`). This can be used to determine exactly which procedure is being called in a call, and allows you to treat the procedure name as a normal atom.