Modes, Determinism, and Mercury

1 Overview

This handout introduces the concept of *modes*, along with their relationship to how many solutions a given procedure produces. Modes allow us to concisely explain what a procedure is expected to take, along with what we can reasonably expect a procedure to do given certain inputs. Modes help us reason about Prolog code, and are often provided in library documentation. However, this sort of reasoning is informal, and it is definitely limited to documentation; nothing in Prolog reasons about modes, nor does the language enforce that the code we write actually agrees with the modes in the documentation.

For this reason, we introduce a new logic programming language, known as *Mercury* (`https://www.mercurylang.org/`). Unlike Prolog, Mercury allows us to write mode information directly in the code itself, and it forces us to comply with what we write. In fact, Mercury *requires* us to specify modes. While this requirement is occasionally burdensome, this is helpful for understanding Mercury code. Additionally, mode annotations allow the Mercury compiler to optimize our code much more heavily than what Prolog allows.

First, we introduce the concept of modes, followed by their relationship to what procedures produce (specifically *determinism*). We finally end with a taste of Mercury, specific with relationship to modes and determinism.

2 Modes

For a moment, let's revisit arithmetic constraints, *before* constraint logic programming was introduced. As previously discussed, these only work correctly when we know the values of any variables involved. For example, consider the queries below:

```
?- 3 < 4.
true.

?- X < 4.
ERROR: </2: Arguments are not sufficiently instantiated

?- X < 4, X = 2.
ERROR: </2: Arguments are not sufficiently instantiated

?- X = 2, X < 4.
X = 2.
```

Up until this point, to say that the inputs to arithmetic constraints like `</2` need to be known was conveyed by saying, well, the inputs need to be known. By relying on English, this ends up being somewhat lengthy to describe, and it's difficult to scale this up to more complex things. For example, consider the following uses of `is/2`:

```
?- X is 2.
X = 2.

?- 2 is 2.
true.

?- 2 is X.
ERROR: is/2: Arguments are not sufficiently instantiated

?- X is Y.
```

```
ERROR: is/2: Arguments are not sufficiently instantiated
```

As shown, while the thing on the left of `is/2` is *usually* a variable, `is/2` still works correctly even if it's not. However, the thing on the right of `is/2` must *not* be a variable, or else we get a runtime error.

The fact that certain inputs need to be known or not for a successful call is captured formally by an idea known as *modes*. Modes are a compact way of annotating procedures to say whether or not an input needs to be known. If you've looked at the official documentation behind any of the SWI-PL library procedures, you've already seen modes, perhaps without knowing what they mean. For example, the official documentation describing `</2` (`http://www.swi-prolog.org/pldoc/doc_for?object=(%3C)/2`) lists the header of the procedure as follows:

```
+Expr1 < +Expr2
```

The `+` is a *mode annotation* saying that the parameter that follows is intended to be an input. Specifically, in this case, both `Expr1` and `Expr2` are intended to be inputs, since they are both preceded by `+`.

The `+` annotation is in contrast to the `-` annotation, which states that something is intended to be an output. For example, the official documentation for `is/2` (`http://www.swi-prolog.org/pldoc/doc_for?object=(is)/2`) lists the header of the procedure as follows:

```
-Number is +Expr
```

Considering the normal usage of `is/2`, the above annotations make sense: it evaluates an arithmetic expression containing no unknown values (`+Expr`) down to a value, and the value is placed on the lefthand side (`-Number`). As shown with the previous queries to `is/2`, it's still ok to put an input in a position where an output is expected; in such a case, the input is simply unified with the output value, behaving more like a check than a computation. (This is the difference between the check `2 is 2` and the computation `X is 2`.)

The last mode annotation frequently used in SWI-PL's documentation is `?`, which states that a given parameter can be *either* an input or an output. For example, consider the header for `append/3` in the official documentation (`http://www.swi-prolog.org/pldoc/doc_for?object=append/3`), shown below:

```
append(?List1, ?List2, ?List1AndList2)
```

The above annotation makes sense considering the many ways in which we can query `append/3`:

```
?- append([1, 2], [3, 4], [1, 2, 3, 4]).
true.

?- append([1, 2], [3, 4], Result).
Result = [1, 2, 3, 4].

?- append(List1, List2, [1, 2, 3, 4]).
List1 = [],
List2 = [1, 2, 3, 4] ;
List1 = [1],
List2 = [2, 3, 4] ;
List1 = [1, 2],
List2 = [3, 4] ;
List1 = [1, 2, 3],
List2 = [4] ;
List1 = [1, 2, 3, 4],
List2 = [] ;
false.

?- append(List1, List2, Result).
List1 = [],
List2 = Result ;
List1 = [_G2135],
Result = [_G2135|List2] ;
List1 = [_G2135, _G2141],
Result = [_G2135, _G2141|List2] ;
...
```

As shown, `append/3` can handle both inputs and outputs in different places, and it works consistently across its various uses.

2.1 Relationship to Determinism

Thinking in terms of inputs and outputs to procedures, the various `append/3` queries above show that we can get different numbers of results depending on whether or not a parameter is used as an input or output. Specifically, we can deduce the following from the above queries, listed in order:

1. `append(+L1, +L2, +L3)` succeeds *exactly once* if `L3` is the result of appending `L1` and `L2` together. The query below shows what happens when this is *not* the case:

   ```
   ?- append([1, 2], [3, 4], [5, 6]).
   false.
   ```

   Given these two queries, we can deduce that `append(+L1, +L2, +L3)` succeeds 0-1 times, for any lists `L1`, `L2`, `L3`.

2. `append(+L1, +L2, -L3)` always succeeds *exactly once*, no matter the values of `L1`, `L2`, and `L3` (assuming `L1` and `L2` are, in fact, lists). Phrased another way, `append(+L1, +L2, -L3)` succeeds 1 time.

3. `append(-L1, -L2, +L3)` always succeeds at least once, and can succeed more times (i.e., there are different possibilities for `L1` and `L2`). Phrased another way, `append(-L1, -L2, +L3)` succeeds 1-N times, for some number $N \geq 1$.

With the sort of reasoning above discussing how many answers are possible, we say this is reasoning about *determinism*. This sort of information related to determinism can be useful in understanding code, particularly when different queries are involved. In fact, the documentation for SWI-PL specifies a formal notation to be used in library documentation to relate modes to the number of possible answers (`http://www.swi-prolog.org/pldoc/man?section=modes`).

2.2 Modes, Determinism, and Prolog

While there is utility in specifying modes and determinism in Prolog, this strictly serves as documentation. Nothing in the language actually enforces anything we write, and similarly there are no guarantees that the modes written by us reflect reality. For example, while there is a formal notation for discussing determinism, this is very rarely used in the SWI-PL libraries. Modes are commonly specified, though there are a number of examples where the modes listed are not entirely accurrate.

For example, consider the `atom_number/2` procedure (`http://www.swi-prolog.org/pldoc/man?predicate=atom_number/2`), which is intended to convert an atom like `'123'` into a a number (i.e., `123`). The `atom_number/2` procedure has the following header:

atom_number(?**Atom**, ?Number)

The above modes say that both `Atom` and `Number` may be outputs. However, the following queries show that this isn't quite true:

```
?- atom_number('123', Number).
Number = 123.

?- atom_number(Atom, 123).
 Atom = '123'.

?- atom_number(Atom, Number).
ERROR: atom_number/2: Arguments are not sufficiently instantiated
```

As shown above, while `atom_number/2` works correctly if *either* parameter is an output, it doesn't work correctly if *both* parameters are outputs. In such a case, we end up getting the same error message that motivated modes in the first place!

The correct mode annotations for `atom_number/2` are instead:

atom_number(+**Atom**, −Number)
atom_number(−**Atom**, +Number)

Note that it's ok if a procedure is associated with multiple modes; this is in fact quite common. However, it's decidedly uncommon for Prolog's documentation to list all the modes.

3 Enter Mercury

With the primary motivation of better handling modes, we now introduce a whole new language: Mercury. Mercury, like Prolog, is a logic programming language, and it intentionally has a very similar syntax to Prolog. However, Mercury offers a **lot** of advantages over Prolog, including:

- Enforcement of mode and determinism annotations
- Statically-enforced types with a relatively expressive type system
- Compilation to machine code
- Sophisticated optimizations
- Abstraction over computation with higher-order procedures and functions, as opposed to Prolog's far more error-prone `call`-based technique
- A simplistic but practical module system

Unlike Prolog, Mercury *requires* us to annotate procedures with applicable modes and determinism, and it will *check* that the code actually satisfies the given mode annotations. If this check doesn't succeed, it's a compile-time error in Mercury. While this is occasionally burdensome, this use of modes enables the Mercury compiler to perform some pretty sophisticated optimizations. In particular, the Mercury compiler will statically reorder conjunctions to avoid as much search as possible. This is something that we need to do ourselves when optimizing Prolog code, but Mercury does this for us. Additionally, Mercury can generate specific variants of the same code which are optimized for different modes, and this all is done automatically behind the scenes.

3.1 Defining Procedures in Mercury

We start our discussion of Mercury with an example defining a basic procedure. Instead of defining a procedure from scratch, we will instead port the following simplistic Prolog procedure to Mercury:

```
blah(5).
```

With such a simplistic procedure, we can focus on the sort of annotations Mercury requires us to write for every procedure. These annotations are unique to Mercury.

> ***Sidenote:*** You may be reading this and thinking "why not start with 'hello world'?". Short version: the most basic "hello world" in Mercury is 7 lines of code, and it requires explaining some of the strangest parts of the language first. (If you really want to start there, you may be interested in the following tutorial-style introduction to the language: `https://www.mercurylang.org/documentation/papers/book.pdf`.)
>
> Your next thought is probably "Wow, 'hello world' is 7 lines long and complicated in this language? Mercury must suck." Ok, maybe. But for a moment, think about what a "hello world" program needs to do:
>
> 1. Start execution from some well-defined point.
> 2. Create some representation of a string.
> 3. Tell the operating system we want to display that string somewhere.
> 4. The operating system performs a sequence of operations that is occassionally indistinguishable from magic.
> 5. The literal state of your observable world is changed, because now you see "hello world" somewhere. (Or maybe you don't, and good luck debugging that one, buddy.)
>
> None of this is exactly simple, particularly when it comes to talking to the operating system and changing the state of the world (the latter part is where things get complex with Mercury). I'd argue that something like "hello world" should be at least *somewhat* complicated for these reasons. I may even be a little leery of languages that allow me to mess with the world so easily. (I may also be deranged.)

First of all, Mercury is a statically-typed language, and furthermore Mercury requires that we annotate the types of all procedures. With this in mind, we'll start the definition of the `blah/1` procedure with its type, like so:

```
:- pred blah(int).
```

The above code states that `blah` is a procedure that takes an `int`, Mercury's representation of an integer. The reserved word `pred` is short for *predicate*, that is, something that returns a Boolean value. In Prolog, every procedure is a predicate, since procedure calls always implicity return Boolean values. Mercury introduces other things which we'll get to later, which is why we explicitly say that `blah` is a predicate.

In addition to types, Mercury requires us to specify modes, along with their determinism. We'll exhaustively list all the modes for `blah` below:

```
1 :- mode blah(in) is semidet.
2 :- mode blah(out) is det.
```

The mode on line 1 states that if the first (and only) parameter to `blah` is an input (`in`), then `blah` will succeed 0-1 times (that is, `blah` is *semideterministic*, written `semidet`). The mode `in` behaves exactly like the mode `+` in Prolog; this is just a different name for the same concept. The mode on line 2 states that if the first (and only) parameter to `blah` is an output (`out`), then `blah` will succeed exactly once (that is, `blah` is deterministic, written `det`). The mode `out` behaves exactly like the mode `-` in Prolog; this is just a different name for the same concept.

For determinism, Mercury gives us the following options:

- `det`: succeeds 1 time
- `semidet`: succeeds 0-1 times
- `multi`: succeeds 1-N times, where $N \geq 1$
- `nondet`: succeeds 0-N times, where $N \geq 1$

These options have the same name and meaning in the SWI-PL documentation; it's just that Mercury makes extensive use of these whereas Prolog tends to omit them.

> ***Sidenote:*** There are more options than what is listed, but they are used rarely and exist for special purposes we won't get into. If you're interested in seeing more, consult Section 6 of the Mercury Language Reference Manual (`https://www.mercurylang.org/information/doc-release/reference_manual.pdf`).

As for the actual implementation of `blah`, it is exactly as it is in Prolog, namely:

```
blah(5).
```

Putting everything together, the complete port of `blah` to Mercury follows:

```
1 :- pred blah(int).
2 :- mode blah(in) is semidet.
3 :- mode blah(out) is det.
4 blah(5).
```

Note that the first mode annotation on line 2 isn't needed; Mercury can figure this one out based on the mode annotation on line 3. We show it here for the sake of demonstration, and having it there doesn't hurt anything.