Declarative programming in Mercury

A guide to Mercury's declarative and operational semantics

MARK BROWN

Version 0.4, March 2023

Contents

Preface

This guide started life as some notes and slides aimed at helping YesLogic developers learn the basic principles of logic programming. Most of the developers had plenty of experience in languages such as Haskell and Rust; the main barrier to learning logic programming was, I think, a lack of familiarity with much of the jargon, as well as much of the folklore. Some help was in order from the developers more experienced in Mercury.

The notes first evolved into an article, and then into the format it currently takes. There's a lot more information that could be added, but publishing it in its current form seems like it would still be of use. YesLogic has generously agreed to its release.

Some additional topics:

- A lot more about types, modes, determinism, and uniqueness.
- Modules and abstract data types.
- Minimal model semantics.
- Many-sorted algebra. In the current version we just map everything down to first-order, since part of the argument for declarative programming is that the models are relatively simple.
- User-defined equality and comparison.
- Partial instantiatedness.
- Bibliographical references.
- ...

There is also room for many more working examples.

Chapter 7 is not yet written, though I consider Lee and Harald's work in this area to be important to draw people's attention to. Hopefully, this will be able to be addressed in a future version.

Mark Brown
March 2023

Chapter 1

Introduction

1.1 Purpose

In the Formal Semantics chapter of the Mercury Language Reference Manual, the declarative semantics of Mercury is given in a single paragraph. Readers with sufficient background in logic programming would find the definition familiar: a predicate calculus theory with a "language" specified by the type declarations in the program, and a set of axioms derived from the "completion" of the program. To readers without that background, however, making sense of this can be challenging for a number of reasons.

The case is similar with the operational semantics, which is defined with reference to "SLDNF resolution". The vast majority of people who know about this topic also already know logic programming, so this is not helpful for those who are learning.

The challenge for readers is particularly difficult since existing resources on the predicate calculus tend to come in two forms:

1. Those that focus on logic as it pertains to logic programming.[1] While these do a good job at connecting the logic with an operational semantics (that is, giving the logic a computational interpretation) there is often relatively little focus on the completion semantics, which is how Mercury is defined.

2. Those that focus on classical logic in its own right.[2] While these generally offer a more complete picture of the logic they do not usually discuss resolution, which is the computational mechanism used in logic programming. In addition, the level of mathematical rigor, while important, can obscure the issues most relevant to logic programming.

This guide aims to bridge the gap between theory and practice. It is intended for programmers who have some knowledge of Mercury and want a deeper understanding, but who are unable to derive much practical information from the resources

[1] For example, Apt, K.R., 1997. *From logic programming to Prolog*. London: Prentice Hall.

[2] For example, https://plato.stanford.edu/entries/logic-classical/.

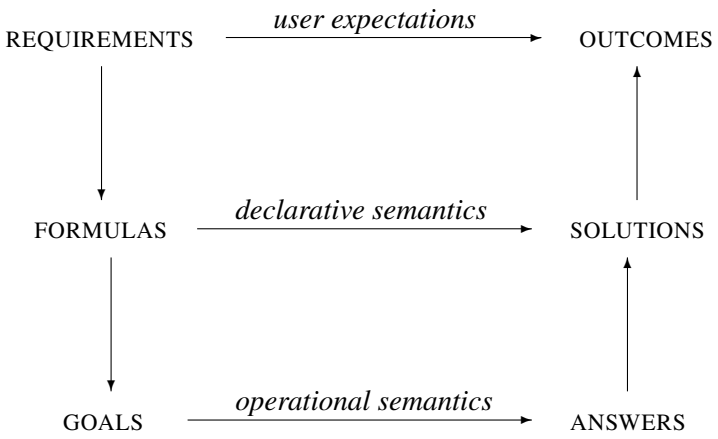

Figure 1.1: Mercury programming in a nutshell.

currently available. It is not presented with the same level of rigor as many other articles on this topic, for example, proofs are not provided for our claims. Rather, the intent is to put programmers in a better position to make the most practical use of existing resources.

1.2 Mercury programming in a nutshell

The main theme of this guide will be to show the parallels between syntax and semantics, of which there are many. By *syntax* we mean the sequences of characters that constitute part or all of a program. The word *semantics* means "meaning", but it also has a technical definition in the context of programming languages, which is that a program semantics constrains the program's behaviour with respect to a particular set of observables.

Figure 1.1 gives a conceptual view of the programming process in Mercury. At a high level, a programmer is given requirements, and in some way or other they need to generate outcomes in accordance with user expectations. They formulate the requirements logically, and with this formulation they can use the declarative semantics to determine what solutions—assignments of values to variables—arise as a consequence. These solutions are then interpreted in terms of the original problem domain, to generate outcomes that (hopefully) satisfy the user.

At a lower level, the programmer's mental formulation is expressed as goals in Mercury. The compiler and runtime system compute answers to the goals in a manner determined by the operational semantics, and these answers can in turn be understood by the programmer as solutions to the formulas.

These two levels of reasoning, the first more abstract and the second more concrete, are represented by the two horizontal arrows in the lower part of the diagram. At each level there is a syntax to express the ideas and a semantics to reason about

what they mean. A close correspondence exists between the two, in that they place the same constraints on a program's observed behaviour.

The difference between the two levels of reasoning, and the reason we would want to consider having two distinct levels in the first place, comes down to how they are defined. The declarative semantics is defined in terms of semantic concepts understood by the programmer, and aims to characterize the programmer's mental picture of how a program behaves. On the other hand, the computer does not have such a mental picture—it blindly manipulates symbols without understanding how the programmer will interpret the results—so the operational semantics aims to characterize the program's behaviour as symbolic manipulation. Thus, the operational semantics is defined in terms of the syntax.

It is the declarative and operational semantics, and the correspondence between them, that is the principal subject of this guide.

1.3 Notation

With the dual syntax vs. semantics view in mind, we adopt a kind of parallel notation and terminology in this guide. When discussing program elements from a primarily syntactic or operational point of view, we will use Mercury syntax written in a monospace font, whereas when the discussion is from a semantic or declarative point of view we will use conventional mathematical notation. Similarly, the terminology used differs between the two sides, with terms that apply to the declarative view having their counterparts in the operational view. Some examples of notation and terminology are show in Figure 1.2.

Hopefully the reader's intuition will be guided by this use of notation. We caution against taking the distinction too seriously, however, as it can sometimes become blurred. Indeed, we will shortly introduce so-called Herbrand interpretations, in which elements of syntax are used directly as elements of the semantic domain.

1.4 Outline of the guide

The outline of the remainder of this guide is as follows.

In Chapter 2 we give an informal picture of the declarative semantics, with a focus on some simple examples. We aim to give a basic idea of what is meant by declarative semantics, and also discuss some of the advantages that can be obtained by thinking about Mercury programs in this way. This provides our motivation for wanting a declarative semantics.

In Chapter 3 we define the syntax and semantics of first-order predicate calculus, and show how the declarative semantics of a Mercury program is expressed in a predicate calculus theory. We give some examples of logical reasoning that can tell us how our programs behave.

In Chapter 4 we give abstract algorithms for unification and resolution, and use these as building blocks to define the operational semantics. We also define the

Semantic/declarative	**Syntactic/operational**
variables:	variables:
x	`X`
$y_1, \ldots, y_n$	`Y1, ..., YN`
values:	ground data terms:
123	`123`
$f(a, g(b))$	`f(a,g(b))`
$a_1, \ldots, a_n$	`a1, ..., aN`
atomic formulas:	atomic goals:
$y = f(x)$	`Y = f(X)`
$p(t_1, \ldots, t_n)$	`p(t1, ..., tN)`
logical connectives:	operators:
$\wedge$	`,`
$\vee$	`;`
$\leftarrow$	`:-`

Figure 1.2: Examples of the parallel notation we will use. The elements themselves will be discussed in later chapters.

negation-as-failure rule that is used to implement negation and if-then-else. Some important results in the meta-theory are given, which we use to show the correspondence between the operational and the declarative viewpoints.

In Chapter 5 we give concrete details of how the implementation behaves at run-time. These details enable programmers to better estimate operational characteristics of their programs, such as stack and heap usage.

In Chapter 6 we extend our work to cover more aspects of the language. In particular, we provide semantics for some Mercury constructs which are not well-characterized in the classical semantics, such as partial functions and exceptions.

In Chapter 7 we present a non-classical interpretation of Mercury programs that is useful for writing specifications and checking that they are correctly implemented. This interpretation demonstrates that classical logic is not the only logic that can be usefully applied to understanding Mercury programs.

Finally, a glossary index in Appendix A provides short definitions, as well as page references, for many of the concepts discussed in the guide.

Chapter 2

Declarative semantics by example

2.1 First examples

Consider the code in Figure 2.1 that defines specialized versions of length and append. A picture of the declarative semantics of `len/1` is given in Figure 2.2. It consists of a table of ground instances (that is, not containing any variables) of the head of the `len/1` function, where the result value is the correct length for the given argument. The table is infinite, but like the multiplication table—which is also infinite—it is relatively easy to get a picture of what is going on by looking at (or thinking about) only a finite part of it.

Similarly, a picture of the declarative semantics of `app/3` is given in Figure 2.3. It again consists of a table of ground instances of the head of the `app/3` predicate, although in this case, since it is a predicate, there is no return value. Instead the arguments must satisfy the relation that holds between two lists and the result of appending them.

Tables like these are known as *Herbrand interpretations*. They assign a meaning to each predicate or function by way of a mapping from ground atoms to truth values: a ground atom is true if it is in the table, otherwise it is false. For example, if the function symbol '+' is interpreted as integer addition then the table will contain entries such as `1 + 1 = 2`. On the other hand, if it is interpreted as string concatenation then the table will contain entries such as `"1" + "1" = "11"`.

Herbrand interpretations are purely syntactic in nature. This reflects the compiler's view of the program: the compiler does not know that `len` is supposed to mean "list length", it only knows it as a symbol. These interpretations also reveal an important fact: the declarative semantics of a program can be understood in its entirety by considering only the truth values taken by the ground atoms That is, there is no need to consider terms that include variables from the program.

Furthermore, this way of thinking scales in the sense that, while we have given examples of two common building blocks, the same applies to much larger pieces of code with far more complexity in their intended interpretations. This ability to scale is crucial for extending our methodology to real-world programs.

```
:- type e ---> a ; b.

:- func len(list(e)) = int.

len([]) = 0.
len([_ | Xs]) = 1 + len(Xs).

:- pred app(list(e), list(e), list(e)).
:- mode app(in, in, out) is det.

app([], Bs, Bs).
app([A | As], Bs, [C | Cs]) :-
    app(As, Bs, Cs).
```

Figure 2.1: Specialized versions of length and append.

```
len([]) = 0                    len([a, a, a]) = 3
len([a]) = 1                   len([a, a, b]) = 3
len([b]) = 1                   len([a, b, a]) = 3
len([a, a]) = 2                len([a, b, b]) = 3
len([a, b]) = 2                len([b, a, a]) = 3
len([b, a]) = 2                len([b, a, b]) = 3
len([b, b]) = 2                ...
```

Figure 2.2: Interpretation of `len/1`.

```
app([], [], [])                app([], [a, b], [a, b])
app([], [a], [a])              app([a], [b], [a, b])
app([a], [], [a])              app([a, b], [], [a, b])
app([], [b], [b])              app([], [b, a], [b, a])
app([b], [], [b])              app([b], [a], [b, a])
app([], [a, a], [a, a])        app([b, a], [], [b, a])
app([a], [a], [a, a])          app([], [b, b], [b, b])
app([a, a], [], [a, a])        ...
```

Figure 2.3: Interpretation of `app/3`.

2.2 Intended interpretations

An interpretation, generally, is something that allows the programmer to comprehend the meaning of terms, and to determine the truth of ground atoms, with reasonable ease. Essentially, it is a specification. If an interpretation reflects what the programmer intends to implement, it is called the *intended interpretation*.

To a programmer, even an informal definition can be sufficient. For example, we could give the intended interpretations of *len/1* and *app/1* as follows:

$$\text{for } n \geqslant 0,\ len([t_1, \ldots, t_n]) = n$$

$$\text{for } n \geqslant 0 \text{ and } 0 \leqslant m \leqslant n,\ app([t_1, \ldots, t_m], [t_{m+1}, \ldots, t_n], [t_1, \ldots, t_n])$$

Although we have used pseudo-code, it should be reasonably clear whether a particular ground atom is true or false in these interpretations. The truth value of goals more generally can be determined by combining the truth values of atomic goals according to the truth tables of classical logic.

It is the usual practice in Mercury to describe the intended interpretation (along with any other pertinent information) in comments immediately preceding a `pred` or `func` declaration. For example, in the Mercury standard library the declarations for the `list.length/1` function and the `list.length/2` predicate appear as follows:

```
    % length(List) = Length:
    % length(List, Length):
    %
    % True iff Length is the length of List, i.e. if
    % List contains Length elements.
    %
:- func length(list(T)) = int.
:- pred length(list(_T), int).
```

Similarly, `list.append/3` is described as follows:

```
    % Standard append predicate:
    % append(Start, End, List) is true iff
    % List is the result of concatenating Start and End.
    %
:- pred append(list(T), list(T), list(T)).
```

It should be easy to see that these are equivalent to the intended interpretations we gave above.

It is a good idea to provide the intended interpretation for any functions or predicates declared in the interface of a module. Doing this enables users of the module to understand whether or not they are using the interface correctly.

We saw above that a list such as `[1,2,3]` can be given the intended interpretation $[1, 2, 3]$. This might seem trivial, but it is worth noting that the former is

meant to represent a piece of Mercury syntax, while the latter is meant to be the kind of notation that might be seen in a semi-formal mathematical proof. Thus it is reasonable to use ellipses and subscripts, or other ad hoc notation, to describe the list structures.

A basic example for lists is the "cons" function, which takes an element and a list, and returns a new list with the element prepended. Since we do not need to know anything about the element we can just call it x, and we can assume the list takes the form $[t_1, \ldots, t_n]$, for some $n \geqslant 0$, and arbitrary elements t_i. We can therefore say that the intended interpretation of cons is a function that, given element x and list $[t_1, \ldots, t_n]$, returns the list $[x, t_1, \ldots, t_n]$.

We have already given intended interpretations for the list append predicate, and for the list length function. The list append function can be specified as:

$$\mathit{append}([s_1, \ldots, s_m], [t_1, \ldots, t_n]) = [s_1, \ldots, s_m, t_1, \ldots, t_n]$$

Similarly, the list reverse function can be specified as:

$$\mathit{reverse}([t_1, \ldots, t_n]) = [t_n, \ldots, t_1]$$

We will use these interpretations later when we implement the queue ADT that is specified in the next section.

2.3 Running example: `queue` ADT

In this section we give the intended interpretation for a (double-ended) queue ADT. We will use this as a running example in the remainder of this guide.

The ADT includes abstract operations to initialize a queue, put elements at the back and get them from the front, unput elements from the back and unget then at the front, and convert the queue to an ordinary list. A queue can be interpreted as a sequence of elements, using the same semi-formal mathematical notation as for lists.

The initialization function, *init/0*, is easy to specify because it just returns (a representation of) the empty queue:

$$\mathit{init} = []$$

Putting an element at the end of the queue is done with the *put/3* predicate, which takes an element, the queue prior to putting the element at the end, and the queue after this has been done:

$$\mathit{put}(x, [t_1, \ldots, t_n], [t_1, \ldots, t_n, x])$$

Getting an element from the front of the queue is done with a *get/3* predicate, which is similar:

$$\mathit{get}(x, [x, t_1, \ldots, t_n], [t_1, \ldots, t_n])$$

```
:- pred append(list(T), list(T), list(T)).
:- mode append(di, di, uo) is det.
:- mode append(in, in, out) is det.
:- mode append(in, in, in) is semidet.    % implied
:- mode append(in, out, in) is semidet.
:- mode append(out, out, in) is multi.
```

Figure 2.4: Declarations for `list.append/3` in the standard library.

The inverse operations, *unput/3* and *unget/3*, are the same as the forward operations except that the second and third arguments are reversed:

$$unput(x, [t_1, \ldots, t_n, x], [t_1, \ldots, t_n])$$
$$unget(x, [t_1, \ldots, t_n], [x, t_1, \ldots, t_n])$$

Finally, the function *list/1*, that converts a queue to an ordinary list, is trivial to specify:

$$list([t_1, \ldots, t_n]) = [t_1, \ldots, t_n]$$

That is, the intended interpretation of *list/1* is the identity function.

When implementing queues according to this specification, the queue type may be defined differently to the list type, though they are both interpreted as sequences of elements. Assuming they are different, the implementation of *list/1* will not be trivial like the specification, but will have to convert between the two representations.

2.4 Purity

2.4.1 Types, modes, and purity

The code we have seen so far is considered *pure*. Code that is written using all but a small number of Mercury constructs is automatically pure—most Mercury code is pure without the programmer needing to think about it. We give the following definition of what exactly we mean by purity.

Definition 1 (Purity)**.** *A predicate or function is* pure *if there exists a declarative interpretation describing its behaviour, that consistently applies in all circumstances.*

In other words, if for a given function or predicate we can picture in our minds a Herbrand interpretation that characterizes the outcome of all possible calls, then that predicate or function is pure.

In the last section we gave the `pred` declaration for `list.append/3` as it appears in the standard library. Figure 2.4 gives this `pred` declaration again, along with all of the declared modes.

The role of the `pred` declaration, which supplies the argument types, should be clear as regards the interpretations we have seen so far: a ground atom that is true in the interpretation will have arguments that are correctly typed, for some values of the type variables. The compiler is able to check that this is the case.

The role of the `mode` declarations is perhaps not so clear in the declarative semantics, but in conjunction with the determinism each one constrains the set of ground atoms that are true, for the predicate or function as a whole. Importantly, and in line with our definition above, the same interpretation applies to *all* of the modes. In Mercury, a predicate or function is pure only if it has a consistent interpretation across all calls, regardless of the mode in which the call is made.

This means, for example, that if a call to *append*([1], [2, 3], z) yields a solution $z = [1, 2, 3]$, which it does, then a call to *append*(x, y, [1, 2, 3]) should at some stage yield the solution $x = [1]$, $y = [2, 3]$, which it also does. Both solutions derive from the fact that *append*([1], [2, 3], [1, 2, 3]) is true in the declarative interpretation. The same applies for any other pair of calls to different modes of *append*.

It should not be surprising that this is the case. The declarative semantics determines which solutions will be produced, and it in turn is determined by the clauses that define a predicate or function. Since the same clauses apply to all modes, no discrepancy can arise.

2.4.2 Mode-dependent clauses

It can be useful to define multi-moded predicates such as `append/3`, the use of which can readily enable larger multi-moded predicates to be implemented. However, it can sometimes be the case that, irrespective of how the clauses defining a predicate are written, making one mode efficient inevitably results in another mode being inefficient. An example of this is the following mode of `append/3`, which is commented out in the standard library.

```
% :- mode append(out, in, in) is semidet.
```

The explanation for why it is commented out says that the mode "is `semidet` in the sense that it does not succeed more than once—but it does create a choice-point, which means both that it is inefficient, and that the compiler can't deduce that it is semidet. Use `remove_suffix` instead."

If such a mode is nonetheless required for a predicate, and the inefficiency would otherwise be unacceptable, then one solution is to implement the predicate using mode-dependent clauses, also known as "different clauses for different modes". Doing this, however, would invalidate our claim from the previous section that the declarative semantics is consistent across modes, since the meaning is no longer determined from a single set of clauses. If the clauses for different modes express different relations, then a consistent declarative semantics cannot exist.

So this is an issue of purity. Having a consistent declarative semantics is required, yet there is no way for the compiler to verify, in general, that two different

sets of clauses express the same relation. As such, the compiler will treat a predicate defined using mode-dependent clauses as impure, unless the programmer is prepared to promise otherwise.

In the Prolog literature, impure predicates—those that do not have a consistent declarative semantics—are typically referred to as "non-logical". (Note that in Section 3.2, we will see this term used in a different sense.) The canonical example of a non-logical predicate in Prolog is `var/1`, which succeeds if and only if the argument is not bound. This can be implemented in Mercury as follows.

```
:- impure pred var(T).
:- mode var(in) is failure.
:- mode var(unused) is det.

var(_::in) :- false.
var(_::unused) :- true.
```

A call to `var(X)` would succeed if, at the point of call, `X` was not bound to anything. This implies that the predicate is interpreted as true for *every* possible argument value. If the same call was made with `X` bound to `a`, however, then the call would fail. This implies that the predicate is interpreted as false for the value *a*, which contradicts the previous implication. Thus it can be seen that the predicate is not pure according to our definition, since there does not exist a declarative interpretation that applies to all modes.

In the next section we give an example, which arose in the course of developing the standard library, of using the declarative semantics to resolve a problem related to the multiple modes of the `string.append/3` predicate.

2.4.3 Case study: `string.append/3`

In early versions of the Mercury, strings were defined as NUL terminated sequences of ASCII characters. With such an arrangement, strings could be regarded as equivalent to lists of characters, and a `string.append/3` predicate could be defined that was analogous to the append operation on lists. In particular, the predicate had a "forward" mode that appended strings, as well as a "reverse" mode that split them apart.

At some point the switch to Unicode was made, which means that strings could no longer be considered sequences of characters. Rather, strings are defined as sequences of code units, which in Unicode are not the same thing as code points (characters). While a sequence of code points, in general, consists of a well-formed sequence of one or more code units, the sequence of code units that constitutes a string is not necessarily well-formed.

In the forward mode of the Unicode version of `string.append/3`, the behaviour is to append the sequences of code units representing the (possibly ill-formed) strings. For backwards compatibility it is desirable to provide the reverse mode that was previously available, but what should such an implementation do,

precisely? At what places should the string be split—between code points where possible, as would make most sense, or between all code units even if in the middle of a code point?

We can use the declarative semantics to help answer these questions. Consider a well-formed string s_3 that has been split into two strings, s_1 and s_2, at a point that is in the middle of a sequence of code units representing one of the code points. Thus, neither s_1 nor s_2 is well-formed, despite the fact that s_3 is.

We would expect the forward mode to append s_1 and s_2 to form s_3. Thus it should be clear that the interpretation of append must include ground atoms such as $append(s_1, s_2, s_3)$, in which the first two arguments are not well-formed, while the third is.

This forces our hand on the reverse mode, however. Since these ground atoms are true in the declarative semantics, the reverse mode must include them in its solutions. In other words, the reverse mode, if it is to be included, cannot just split between code points, it must also split between code units that comprise a single code point. This is true even if the string being split is well-formed.

A reverse mode of `string.append/3` in Unicode would therefore be fundamentally different from the equivalent in ASCII, in which a well-formed string (that is, not containing any NUL characters) could never be split into strings that are ill-formed.

In the end this was considered too different from the original intent of the predicate, and too likely to subtly break code that relies—unwisely, given the possibility in Unicode of ill-formed strings—on the reverse mode only splitting between code points. Thus removing the mode and putting that functionality into a separate predicate was deemed the best solution, despite the fact that it would cause the compiler to issue an error until affected code was updated.

2.5 Partial correctness

Clauses are supposed to state things that are true in the intended interpretation. For example, in Figure 2.1 the first clause of `len/1` is a fact that states that the length of the empty list is zero. The second is a fact that states that, no matter what expressions we substitute for the variables `_` and `Xs`, the length of `[_ | Xs]` will be one greater than the length of `Xs`—in other words the length of a non-empty list is one greater than the length of its tail. In each instance the statements are true according to our intended interpretation.

Furthermore, the clauses cover every possible list, in the sense that every list is either empty or non-empty, and every non-empty list has a tail that is also a list. Perhaps surprisingly, this is enough to conclude that our implementation is correct, at least as far as arguments and return values are concerned.

Our argument here depends on two points, which may be regarded as two sides of the same coin:

- Each of the clauses defining the function is a valid statement, where by valid

we mean that every instance of the clause is true in the intended interpretation. This ensures that there are no "wrong answers", which are ground atoms that are true according to the program as written, but are not intended to be true. We refer to this condition as *clause soundness*.

- Between them, the clauses cover every possible ground atom that is true in the intended interpretation. This ensures that there are no "missing answers", which are ground atoms that are false according to the program as written, but are not intended to be false. We refer to this condition as *clause completeness*.

The two classes of bugs being avoided here, wrong answers and missing answers, are the bugs that are observable in the (classical) declarative semantics.

It is worth noting that, in many cases, Mercury's mode and determinism systems can make the compiler check the coverage for us: if the determinism indicates that calls cannot fail (that is, if the determinism is `det`, `multi`, or `cc_multi`), then all possible values of the type must be covered for any argument with mode `in`. In this case the `len/1` function is `det` and the argument is an input, because the default function mode is being used. Had we not covered every possible list, the compiler would have issued a determinism error, so we can safely assume that the definition covers all possible solutions. The same also applies to any other function declared with the default mode.

Continuing with the examples, the first clause of `app/3` is a fact that states that if you append the empty list and any other list, the result will be the same as the other list. The second clause is a rule; these are taken as logical implications in which the body implies the head (that is, `:-` is interpreted as $\leftarrow$). So this is stating that, for any variable assignment, if `Cs` is the result of appending `As` and `Bs`, then `[X|Cs]` is the result of appending `[X|As]` and `Bs`. Again, both clauses are valid according to the intended interpretation, and the definition is clause complete. In this case clause completeness means that, for every atom that is intended to be true, there is either a fact that covers it or a rule whose head covers it and (under the same variable assignment) whose body is intended to be true.

Since both conditions are satisfied, we can conclude in a similar way to `len/1` that our implementation of `app/3` is correct.

We have been saying "correct" here, but this is only as far as the arguments (and return values, if present) are concerned, which is to say that there are neither wrong nor missing answers. Other kinds of bugs are not observable in the declarative semantics, such as unintended exceptions or nontermination, poor computational complexity, or unbounded stack usage. We therefore refer to correctness in the above sense as *partial correctness*. Notionally this is akin to type correctness, in the sense that it rules out a certain class of bugs but cannot rule out all bugs.

The results of this section can be summarized with a theorem.

Theorem 1 (Partial correctness)**.** *If every clause in a program is true in the intended interpretation and every predicate and function definition is clause complete according to this interpretation, then the program is partially correct.*

```
:- type integer == list(int).

:- func to_string(integer) = string.
to_string([]) = "0".
to_string(As @ [_ | _]) =
    append_list(map(string, reverse(As))).

:- func add(integer, integer) = integer.
add([], Bs) = Bs.
add(As @ [_ | _], []) = As.
add([A | As], [B | Bs]) = [A + B | add(As, Bs)].
```

Figure 2.5: A buggy implementation of arbitrary precision integers.

A corollary of this is that if a program is not partially correct, that is, if it produces wrong answers or misses answers that it should have produced, then there is at least one clause in the program with an instance that is not true in the intended interpretation, or at least one definition that is not clause complete.

2.6 Declarative debugging

The process of debugging, broadly speaking, can be broken down into the following three phases.

1. Observing a bug symptom. For declarative debugging, the symptoms of interest are wrong answers and missing answers.

2. Bug localization. For declarative debugging, we can narrow the immediate source of a bug down to a single clause or definition. Theorem 1 implies that this is possible.

3. Bug fixing. This is beyond the scope of what is usually called declarative debugging, but understanding the code in terms of the intended interpretation can still help with actually fixing the bug.

It is in the second phase, bug localization, that declarative debugging is particularly effective.

Consider the snippet of (buggy) code in Figure 2.5, that represents arbitrary precision integers as lists of decimal digits starting from the least significant. It provides functions to convert an integer to a string and to add integers.

If we call `to_string(add([2],[5,1]))` the answer `"17"` is returned, which is the answer that we intended. If we call `to_string(add([6],[7]))` it returns `"13"`, which is also the answer that we intended—in this case there were incorrect intermediate values, but we did not observe this in the result so we are not in a

position to commence debugging. If we call `to_string(add([2,6,1],[3,7]))` then we intend it to return `"235"` but instead it returns `"1135"`, which is a wrong answer. Thus we have observed a bug symptom.

To localize this bug we start at the bug symptom, which for us is the atom that was incorrect:

```
to_string(add([2,6,1],[3,7])) = "1135"
```

We can first check the call to `add/2`. In this case the following atom appears:

```
add([2,6,1],[3,7]) = [5,13,1]
```

This is false in the intended interpretation, because integers are supposed to consist of decimal digits in the range 0 to 9. Since this is a wrong answer, we proceed to debug the atom. It matches the third clause of `add/2`, the instance of which contains the following calls:

```
2 + 3 = 5
add([6,1],[7]) = [13,1]
```

The first is obviously true, but the second is false because $16 + 7 = 23$, so we intended the result to be `[3,2]`. Again, the third clause is matched and the calls made are:

```
6 + 7 = 13
add([1],[]) = [1]
```

In this case *both* atoms are true in the intended interpretation. Since in the third clause of `add/2` the result is wrong, and since none of the calls it made showed any symptoms, we can conclude that the third clause contains a bug. And indeed it does, since it does not allow for a carry bit to flow over to the next element.

The transcript of an `mdb` session, in which the declarative debugger is used to algorithmically debug the same problem as above, is shown in Figure 2.6. The process looks a bit different to how we just described it, but that is because the debugger assumes it does not need to ask about the correctness of `+`, for example.

Bug fixing, the third phase of debugging, might start with defining a function `add_with_carry/3`, where the intended interpretation is:

$$\text{for } a, b \in \mathbb{Z},\ c \in \{0, 1\},\ \mathit{add\_with\_carry}(a, b, c) = a + b + c$$

Then `add/2` could be implemented as:

```
add(As, Bs) = add_with_carry(As, Bs, 0).
```

Implementing the function `add_with_carry/3` is left as an exercise.

```
Melbourne Mercury Debugger, mdb version DEV.
Copyright 1998-2012 The University of Melbourne.
Copyright 2013-2023 The Mercury team.
mdb is free software; there is absolutely no warranty...
       1:      1  1 CALL pred arbint.main/2-0 (det)
mdb>
       2:      2  2 CALL func arbint.add/2-0 (det)
mdb> f
      17:      2  2 EXIT func arbint.add/2-0 (det)
mdb> dd
add([2, 6, 1], [3, 7]) = [5, 13, 1]
Valid? n
add([6, 1], [7]) = [13, 1]
Valid? n
add([1], []) = [1]
Valid? y
Found incorrect contour:
+(6, 7) = 13
add([1], []) = [1]
add([6, 1], [7]) = [13, 1]
Is this a bug? y
      16:      4  3 EXIT func arbint.add/2-0 (det)
mdb> quit -y
```

Figure 2.6: An `mdb` declarative debugging session.

2.7 Summary

Interpretations of function and predicate symbols provide us with a meaning for our programs. Herbrand interpretations give the meaning from the compiler's point of view, in purely syntactic terms. More generally, interpretations describe, sometimes informally, the behaviour of the functions and predicates in a program with regard to its inputs and outputs.

The interpretation of multi-moded predicates greatly helps in clarifying how different modes should behave. This is particularly important when considering the use of mode-dependent clauses, such as with the standard library implementation of `string.append/3`, for which the compiler cannot completely verify logical consistency.

One interpretation in particular is called the intended interpretation. This acts as a specification that reflects what the programmer intends to implement. With it, we can determine the partial correctness of a program, which allows us to rule out certain classes of bugs.

In particular, it is expressive enough to allow us to determine whether an individual clause is sound, and whether a definition is complete. Thus, it can assist with bug localization and fixing, as well as help with avoiding bugs in the first place. We have given some simple examples to illustrate this, as well as introducing a queue ADT which we will use as a running example.

In the remainder of this guide we take a more formal look at how Mercury's semantics is defined, as well as some additional topics of interest.

Chapter 3

First-order predicate calculus

3.1 Overview

In this chapter we introduce first-order predicate calculus, also known as first-order logic, or classical logic. This mathematical notion forms the basis of Mercury's declarative semantics, via the translation of Mercury code into axioms of a predicate calculus theory.

Articles on first-order predicate calculus typically define a syntax, then introduce a deductive system that allows proofs to be constructed from inference rules, then define a semantics in terms of "interpretations" and "models" and prove some results in the meta-theory. We will take a similar approach, however we present the semantics, with a particular focus on how (first-order) Mercury programs are converted to logic formulas, before the deductive system is covered. In the next chapter we will present the deductive system, which gives a computational interpretation to the logic.

Mercury supports higher-order code, of course, so a first-order description will not directly cover all of it. First-order logic is also untyped. Despite this, we can reduce Mercury's declarative semantics to first-order logic, which is conceptually much simpler than the alternatives—first-order theories have relatively simple models, such as the Herbrand interpretations from the previous chapter—so that is the approach we will take in this guide. We will discuss types and higher-order code, plus some other extensions of the basic logic, in Chapter 6.

In the remainder of this chapter we give the syntax of predicate calculus, and show how basic Mercury constructs generate formulas, and how Mercury programs generate axioms. We then give a formal definition of the classical semantics that results from those axioms. Some examples of ad hoc proofs based on the semantics are provided, to help illustrate the concepts introduced, and to motivate the operational semantics that we define in detail in the next chapter. We conclude the chapter with some philosophical remarks about the role of classical logic in understanding computer programs.

1. An infinite set of variables: x, y, z, ...
2. Logical constants: *true*, *false*
3. Logical connectives: $\wedge$, $\vee$, $\neg$, $\leftarrow$, $\rightarrow$, $\leftrightarrow$, ...
4. Quantifiers: $\forall$, $\exists$
5. Logical relations: $=$
6. Punctuation: we will use parentheses and commas in the usual way.
7. For $n \geqslant 0$, a set of function symbols with arity n: $f/2$, $g/1$, ...
8. For $n \geqslant 0$, a set of predicate symbols with arity n: $p/2$, $q/3$, ...

Figure 3.1: Symbols used in predicate calculus.

3.2 Syntax

The syntax of first-order logic is given in terms of a set of symbols, along with rules to say how they can be put together to make well-formed terms and formulas. The symbols we use are listed in Figure 3.1.

Some sources refer to the symbols in categories 1 to 6 as *logical*, while referring to the symbols in categories 7 and 8 as *non-logical*. It is important to note that this term is being used in a different sense to that in Section 2.4. In this context it is referring to whether symbols are defined as part of the logic itself, or defined as part of a particular theory. To put it in programming terms, "logical" here means that the symbol is a fixed part of the language, while "non-logical" really just means user-defined.

By convention, we will use the names v, w, x, y and z, possibly with subscripts, to stand for arbitrary variables. A sequence of variables may be written with an overbar, as in $\bar{x}$. The logical constants, logical connectives and quantifiers have their usual meanings, and we will write formulas following the usual rules of operator precedence. Parentheses will be used in the conventional way to override this when required.

The non-logical symbols are derived from declarations in the Mercury program. Function symbols include the data constructors declared in discriminated union types, as well as the declared functions. Predicate symbols are the declared predicates. We will use names such as f and g to stand for arbitrary functions, and names such as p, q and r to stand for arbitrary predicates. We will sometimes refer to function symbols with arity zero as *constants*, and use names such as a, b and c to stand for arbitrary constants.

The arity of a predicate or function symbol is listed after a slash, for example $f/2$ or $p/1$. However, we will sometimes leave the arity off entirely when it is zero

Terms:

t	$::=$	x	variable
	$\mid$	a	constant
	$\mid$	$f(t_1, \ldots, t_n)$	compound term

Formulas:

ϕ	$::=$	*true*	always true
	$\mid$	*false*	always false
	$\mid$	$t_1 = t_2$	equation
	$\mid$	$p(t_1, \ldots, t_n)$	predicate call
	$\mid$	$\phi_1 \wedge \ldots \wedge \phi_n$	conjunction
	$\mid$	$\phi_1 \vee \ldots \vee \phi_n$	disjunction
	$\mid$	$\neg\phi_1$	negation
	$\mid$	$\forall x.\, \phi_1$	universal quantification
	$\mid$	$\exists x.\, \phi_1$	existential quantification

Figure 3.2: The grammar rules for terms and formulas.

or clear from the context. Note that the arity is part of each symbol's identity: two symbols with the same name but different arities are considered different symbols.

The grammar rules for the predicate calculus are shown in Figure 3.2. Terms are constructed from variables and function symbols in essentially the same way that expressions are in Mercury. We will refer to arbitrary terms using names such as s and t. Terms that are constructed using only variables and data constructors (that is, with no function calls) play an important role in our discussion; we will refer to these as *data terms*. A data term that contains no variables we will refer to as a *ground data term*.

We will assume the existence of two function symbols, $[|]/2$ and $[]/0$, representing the list constructor and the empty list, respectively. We will write list terms in an analogous way to Mercury list syntax, for example, we will write $[1, 2, 3]$ as a shorthand for $[|](1, [|](2, [|](3, [])))$.

Formulas are either atomic or compound. Atomic formulas are either logical constants, equations, or predicate calls. Compound formulas are constructed from atomic formulas using connectives and quantifiers, in essentially the same way that goals are in Mercury. We will refer to arbitrary formulas using names such as ϕ and ψ.

Quantifiers with multiple variables stand for the same quantifier with each variable in turn, that is, $\forall xy.\phi$ is an abbreviation for $\forall x.\forall y.\phi$. Also, the scope of a quantifier extends as far as possible; parentheses will be used if the scope needs to be limited.

A variable occuring in a formula is *bound* if it is captured by a quantifier, otherwise it is *free*. For example, x is bound and y is free in the formula $\forall x.\, f(x, y) = y$. (Despite the nomenclature, these notions are different from the notions of bound

Expressions ⇒ Terms:

`X`	x
`a`	a
`f(t1, ..., tN)`	$f(t_1, \ldots, t_n)$

Goals ⇒ Formulas:

`true`	*true*
`false`	*false*
`t1 = t2`	$t_1 = t_2$
`p(t1, ..., tN)`	$p(t_1, \ldots, t_n)$
`Goal1, ..., GoalN`	$\phi_1 \wedge \ldots \wedge \phi_n$
`( Goal1 ; ... ; GoalN )`	$\phi_1 \vee \ldots \vee \phi_n$
`( if C then T else E )`	$(\phi_c \wedge \phi_t) \vee (\neg\phi_c \wedge \phi_e)$
`some [X] Goal`	$\exists x.\, \phi$

where ϕ is the formula corresponding to `Goal`

Figure 3.3: Semantics of Mercury expressions and goals.

and free in Mercury's mode system.)

A formula that has no free variables is said to be *closed*; some sources refer to closed formulas as "sentences". The universal closure of a formula is that formula with all of its free variables universally quantified.

3.3 Expressions and goals

One of the building blocks for understanding Mercury's declarative semantics is to see how Mercury expressions are mapped to terms, and how Mercury goals are mapped to formulas. The relationship we want to describe here is the one represented by the lower left vertical arrow in Figure 1.1 on page 2.

The mappings for expressions and goals are provided in full detail in the reference manual, but for convenience Figure 3.3 summarizes the most important bits. In line with our notational convention, ϕ is the formula that corresponds to goals such as `Goal`. A subscript may be used in cases where there are multiple goals, so `Goal1` maps to ϕ_1, and so on.

Negations are not listed as goals, since in Mercury a negated goal is an abbreviation for a conditional goal. Specifically, the following translation takes place at the level of Mercury syntax.

```
not G    ⇒    ( if G then false else true )
```

It should be easy to see that the resulting formula is equivalent to negation, since the goal on the right maps to the formula

$$(\phi \wedge \mathit{false}) \vee (\neg\phi \wedge \mathit{true})$$

which is logically equivalent to $\neg\phi$.

Similarly, universal quantifications are not listed as goals. In this guide, the universal quantifications we encounter will be generated in ways other than from goals. The reference manual specifies all cases in which goal constructs are actually just syntactic abbreviations.

The mapping we have given here is not quite the full story, since we will need to add implicit quantifiers to the formulas, and extend the mapping to program clauses. We will address these points in the next two sections.

3.4 Implicit quantification

The usual rule in mathematics is that free variables in a formula are implicitly universally quantified across the whole formula.[1] This is not quite what we want for Mercury's semantics, since in Mercury code there are common cases which require existential quantification in order to avoid spurious errors. It is convenient for the programmer, and produces a natural result, if these existential quantifications are added implicitly before applying the usual mathematical rule.

In first-order code, the most important case is that of variables in a conditional goal that are used in the condition and possibly also in the then-branch, but not anywhere else that bindings from the condition could reach during execution. For a conditional that maps to the formula $(\phi_c \wedge \phi_t) \vee (\neg\phi_c \wedge \phi_e)$, if $\bar{x}$ is the set of variables in question then the formula is implicitly quantified as follows.

$$(\exists \bar{x}.\, \phi_c \wedge \phi_t) \vee (\neg(\exists \bar{x}.\, \phi_c) \wedge \phi_e)$$

Note that the first quantifier ranges over the then-branch but the second quantifier does not range over the else-branch. This reflects the fact that variables bound in the condition can be used in the former but not the latter.

The implicit quantification process is applied *after* the expansion of syntactic abbreviations, so an analogous process effectively applies to negations and other goals that are abbreviations for conditional goals.

3.5 Clauses

To give a semantics to clauses, we can consider a mapping that just replaces `:-` with reverse implication. Figure 3.4 shows the effect this has on the different forms of clauses—the formulas that result are implications in which the body implies the head, which is the interpretation discussed in Section 2.5. This essentially lines up with clause soundness, in that if one of these implications is incorrect it will lead to a wrong answer bug.

[1] For example, in a mathematical identity such as $\sin^2\theta + \cos^2\theta = 1$, the equation is meant to be taken as true for *all* values of θ.

Clauses ⇒ Formulas:

`p(t1, ..., tN) :- Body`	$p(t_1, \ldots, t_n) \leftarrow \phi$
`p(t1, ..., tN)`	$p(t_1, \ldots, t_n) \leftarrow true$
`f(t1, ..., tN) = t :- Body`	$f(t_1, \ldots, t_n) = t \leftarrow \phi$
`f(t1, ..., tN) = t`	$f(t_1, \ldots, t_n) = t \leftarrow true$

where ϕ is the formula corresponding to `Body`, if present

Figure 3.4: Mercury clauses as reverse implicatios.

Two problems still remain with this formulation of clauses. First, the resulting formulas are not necessarily closed. We will see shortly that we require the formulas that go into our semantics to be closed. Second, while we have made the connection to clause soundness, we have not yet done so for clause completeness. That will require looking at all clauses that define a predicate or function, instead of just one at a time.

Before we continue our discussion of clauses, however, we will first need to take a closer look at equality as it relates to the predicate calculus. While the topic is usually taken as basic in logic, there are slightly different approaches and it is worth looking closely at the choices we have made.

3.6 First-order logic with equality

In our syntax we included equality as one of the logical symbols. We define this as *semantic* equality: two terms are equal if and only if they denote the same thing. Thus, by definition, the relation is reflexive, symmetric, and transitive, as would be expected. A substitutivity principle also applies, in that if any argument to a function is replaced by another argument that is equal according to our definition, then the result is equal. Similarly, if any argument to a predicate is replaced by another equal argument, then the result is logically equivalent.

Another possible definition of equality, which we do not adopt, is *syntactic* equality. This means that two terms are considered equal if they are syntactically identical, or if they can be made so via a variable substitution. The two definitions are very similar, and produce the same results in most cases. There are subtle differences, however, and these will eventually become relevant when we talk about `semidet` functions in Section 6.2, so it important to take note of precisely which definition we are using.

When the equality relation is defined as part of the logic, as we have done in this guide, the logic is sometimes called *first-order logic with equality*. Some authors take an alternative approach where the equality relation is treated no differently to other predicate symbols. This alternative approach is sometimes used in order to define equality in terms of some other relation, for example by saying that two sets are equal if each is a subset of the other. In typical cases, however, equality is taken

to be part of the logic, and the term "first-order logic" is usually taken to mean first-order logic with equality.

We have chosen to include equality as part of the logic since it is simpler to do so, and we do intend the aforementioned properties to hold. This does not capture our full intent, however, and we will need to add to our definition in the next section.

3.7 Axioms

3.7.1 What are axioms?

Axioms are closed formulas that are taken to be true without proof, meaning they may be used as a starting point for proofs of other formulas. We will look more at proofs in the next chapter. A closed formula that can be proven from a set of axioms is called a *theorem*, and the collection of all such closed formulas is known as a *theory*. Thus, the axioms form the basis of a particular predicate calculus theory.

In this section we will show how to generate a set of axioms from the declarations and clauses in a Mercury program. The theory that results will ultimately determine the declarative semantics of the program. We will make this concept clearer once we have discussed models, which we will do in Section 3.8.

Note that in some sources on this topic the authors' aim is to axiomatize the logic itself—that is, include axioms that define the logical symbols—but we take the approach of defining the logic symbols directly and just using axioms to define what is specific to a program. This is because we want to state something *using* the logic, rather than explore theorems *about* the logic. Thus, when we refer to "axioms" we mean those that are generated from the program, rather than the sort that define the logic being used.

3.7.2 Equality axioms

Our definition of equality thus far gives us some conditions that imply when two given terms are equal. It does not, however, say when they are not equal. As it stands, we are not even able to rule out the case where *all* terms are equal. To repair this, we will need to add some axioms relating to equality. Our intent is that two ground data terms (that is, terms that do not include function calls) should be equal only if they are syntactically identical.

Note that we are saying *only if*. In other words, equality of ground data terms implies that they must be syntactically identical. This is not the same as saying syntactic identity of ground data terms implies that they must be equal, since the implication is in the opposite direction. If the implication were this way around we would be defining equality as syntactic, but, as mentioned in Section 3.6, that is not the definition of equality we have adopted.

The condition that ground terms are equal only if identical is known as the *Unique Names Assumption*. As the name suggests, this is often left implicit, par-

```
:- type foo ---> a ; b.
:- type bar ---> nil ; cons(foo, bar).
```

Figure 3.5: Type definitions for the `foo` and `bar` types.

$$\neg(a = b) \qquad \forall xy.\, \neg(nil = cons(x, y))$$

$$\forall x_1 x_2 y_1 y_2.\, (cons(x_1, y_1) = cons(x_2, y_2)) \rightarrow (x_1 = x_2 \wedge y_1 = y_2)$$

$$\forall x.\, \neg(x = cons(t_1, t_2)), \text{ if } x \text{ occurs in } t_1 \text{ or } t_2$$

Figure 3.6: Equality axioms for the `foo` and `bar` types.

ticularly when discussing logic programming or databases. In other cases, such as when dealing with ontologies, it can be desirable to have two distinct names denote the same thing, in which case the assumption is not made. In our case, there are of course terms containing function calls that are intended to be equal even if they are not syntactically identical. For example $2 + 2$ is equal to $1 + 3$ since both sides are equal to 4, so we do not want this assumption to always apply. We do intend that this assumption holds for ground data terms, however.

Another assumption we wish to make is that all terms are finite. For an equation such as $x = f(x)$, where f is a data constructor, the solution if it exists must be the infinite term:

$$x = f(f(f(f(\ldots))))$$

This is not a term that can be written down using our syntax, however, so we wish to exclude it as a possible solution.

To support these requirements, a number of axioms are generated for each type. Consider the type definitions in Figure 3.5. The constants are a, b and nil, and $cons/2$ is the only other function symbol. Note that *foo* and *bar* are not included, as these are type constructors not data constructors or declared functions.

The equality axioms for `foo` and `bar` are shown in Figure 3.6. The two in the first row say that ground data terms are not equal unless their principal functors are equal. We have omitted any axioms that say terms of different types are not equal, since in general there is a large number of these, and in a type correct program such axioms will not make any real difference. In a non-typechecked setting, all $O(n^2)$ axioms would be required.

The third axiom says that ground data terms are not equal unless their arguments are equal. In other words, this simply states that $cons/2$ denotes an injective function.

The last line is an axiom schema (that is, an infinite family of axioms), where there is one axiom for each pair of data terms t_1 and t_2. This schema is known as the

occurs check,[2] and it states that a variable is never equal to a data term that contains that variable. Equivalently, a data term never strictly contains itself as a subterm. This implies that all data terms are finite; for example the formula $x = cons(a, x)$, which would otherwise describe an infinite term, is always false because x occurs in the term that it is supposedly equal to.

With the axioms we have defined we can no longer say that ground data terms that are syntactically different may be equal. For example, the axioms allow us to infer

$$\neg(cons(a, nil) = cons(b, nil))$$

since if we assume that $cons(a, nil)$ and $cons(b, nil)$ are equal, then by the third axiom we have that $a = b$. This contradicts the first axiom, which means that our assumption must have been false.

Without the equality axioms there would have been no way to derive this proof.

3.7.3 Clause soundness axioms

In Section 2.5 we gave the clause soundness condition for partial correctness, which required that each clause in the program must be true in the intended interpretation. This can be expressed as an axiom in a straightforward way.

Recall from Section 3.5 that a clause for a predicate p with arity n can be mapped to a formula $\psi \leftarrow \phi$, where ψ corresponds to the clause head and takes the form $p(t_1, \ldots, t_n)$, for argument terms $t_1, \ldots, t_n$, and ϕ is the formula corresponding to the clause body (or *true* if the clause is a fact). Similarly for a function f with arity n, except that ψ takes the form $f(t_1, \ldots, t_n) = t_r$, where t_r is the return expression.

The formula can be made into an axiom in the same way mathematical formulas usually are, by taking the universal closure as follows:

$$\forall \bar{x}.\, \psi \leftarrow \phi$$

where $\bar{x}$ is the set of free variables that occur in ψ and ϕ.

The resulting formula is known as the *clause soundness axiom* for the clause. As suggested above, it expresses the clause soundness condition for the clause, and it must be true in the intended interpretation. Since it is an implication, the only way it can be false is if ψ is true and ϕ is false. If this is the case—that is, the axiom is false in the intended interpretation—then, in line with the discussion in Section 2.5, the clause is a wrong answer bug.

Observe that, while for predicates the axiom asserts something about ground atoms for the predicate itself, for functions the axiom asserts something about the equality relation. Unlike data constructors, we do not generate axioms that say the function is injective, or that the function returns values that are different from every other function. The function completion adds new instances of terms being equal, rather than excluding such instances.

[2] Or sometimes "occur check", without the plural.

The axiom also tells us a way to make inferences about the program. We can choose any way we like of assigning values to the variables $\bar{x}$, and if we do so in a way that ϕ, the clause body, is something we have already established to be true, then we can infer that ψ, the clause head, will also be true. Equivalently, if we want to know whether or not an instance of ψ is true, it is sufficient to show that ϕ is true under the same variable assignment.

3.7.4 Combined clause soundness axioms

In the form given in the previous section, the clause soundness axiom is convenient for reasoning about a program one clause at a time. It is also useful, however, to be able to reason about the combination of all clauses that make up a predicate or function definition.

One way to express a combined clause soundness axiom is to directly conjoin all of the individual axioms, as follows:

$$\forall \bar{x}_1 . \psi_1 \leftarrow \phi_1 \wedge \ldots \wedge \forall \bar{x}_m . \psi_m \leftarrow \phi_m$$

where m is the number of clauses in the definition. This does not provide us with much additional insight as it stands, since there are still just as many implications to consider. If, however, the clause heads all have identical arguments, we could easily combine all the conjuncts into a single implication.

We can make the clause heads take the required form by performing a simple logical transformation. For a predicate p with arity n, let $v_1, \ldots, v_n$ be a sequence of variables that are distinct from any other variables in the clauses. We define the following formulas:

$$\begin{aligned} \psi' &\equiv p(v_1, \ldots, v_n) \\ \phi' &\equiv \exists \bar{x} . v_1 = t_1 \wedge \ldots \wedge v_n = t_n \wedge \phi \end{aligned}$$

where $\bar{x}$ is the set of free variables that occur in ψ and ϕ, and $t_1, \ldots, t_n$ are the original argument terms. In other words, ψ' is ψ with the fresh variables in place of the argument terms, and ϕ' is ϕ conjoined with equations between each of the fresh variables and the corresponding argument terms, and with all variables except the fresh ones existentially quantified.

Analogous definitions can be given for functions, with the difference being that we also need a variable, v_r, for the function result, and an additional equation to bind it. In this case we obtain:

$$\begin{aligned} \psi' &\equiv f(v_1, \ldots, v_n) = v_r \\ \phi' &\equiv \exists \bar{x} . v_1 = t_1 \wedge \ldots \wedge v_n = t_n \wedge v_r = t_r \wedge \phi \end{aligned}$$

where $\bar{x}$ and $t_1, \ldots, t_n$ are as before, and t_r is the return expression for the clause.

Now consider the formula $\psi' \leftarrow \phi'$. If we take the universal closure in the same way as in the previous section, the result is logically equivalent to the clause

soundness axiom. Intuitively, moving argument terms into the body via equations with fresh variables does not change the meaning of a clause. The reason we existentially quantify variables in the original clause is because we are only interested in the values of head variables, and because when we combine the clauses, as we will do in a moment, we do not want variables from different clauses to clash if they happen to have the same name. Essentially, the existential quantification reflects that the scope of variables in a clause is just that single clause.

From the collection of clauses in a definition, we obtain ψ', which is common to all the clauses, and which is implied by ϕ'_i for the ith clause. The conjunction of the formulas is found by taking the disjunction of the antecedents:

$$\psi' \leftarrow \phi'_1 \vee \ldots \vee \phi'_m$$

Universally quantifying the free variables, $\bar{v}$, would give us something equivalent to the conjunction of all of the clause soundness axioms for the definition.

3.7.5 Clause completeness axioms

Our definition of partial correctness from Section 2.5 requires not just clause soundness but also clause completeness. That is, the set of clauses defining a predicate or function must, between them, cover every possible ground atom that is true in the intended interpretation.

The *clause completeness axiom*, which expresses the clause completeness condition, is the counterpart to the clause soundness axioms. It can be obtained from the combined clause soundness axiom from the previous section, by changing the reverse implication into a forward implication prior to universally quantifying.

For a predicate or function defined by m clauses, the clause completeness axiom is therefore:

$$\forall \bar{v}. \psi' \rightarrow \phi'_1 \vee \ldots \vee \phi'_m$$

Like the clause soundness axioms, it must be true in the intended interpretation. In this case, the only way it can be false is if the left-hand side is true and the right-hand side is false. In line with the discussion in Section 2.5, if this happens then the definition contains a missing answer bug.

Intuitively, putting the clauses into a disjunction reflects the fact that execution can choose any one of the clauses, and using a forward implication means that, while each clause says what things are true, these are the *only* things that are true. That is, every ground atom that is true must be covered by one of the clauses, as we expect.

We can use the clause completeness axiom to make inferences about the program, in much the same way as we can with the clause soundness axiom. The difference is that it gives us *negative* information—it allows us to infer that a particular ground atom must be false, rather than true. This in turn allows us to reason about whether the negation of a ground atom is true, or about which branch will be taken by a conditional goal.

3.7.6 Predicate and function completion

In most sources, the clause soundness axioms and the clause compleness axiom are combined into a single formula. For a given predicate or function, this formula, which is logiclly equivalent to the conjunction of the axioms, is known as the *completion* of the predicate or function.

Since we already have the axioms in the form of implications in one direction or the other, the conjunction of them is just a bi-implication between the two sides. In other words, the formula is:

$$\forall \bar{v}.\, \psi' \leftrightarrow \phi'_1 \vee \ldots \vee \phi'_m$$

where $\bar{v}$ is the set of fresh variables we introduced.

An example should help to illustrate, and for this we will turn once again to `append/3`. The clauses, when mapped to the $\psi \leftarrow \phi$ form, are as follows (variable names are chosen to fit with our notational convention):

$$\begin{aligned} &\mathit{append}([], y, y) \leftarrow \mathit{true} \\ &\mathit{append}([w|x], y, [w|z]) \leftarrow \mathit{append}(x, y, z) \end{aligned}$$

We obtain the clause soundness axioms by universally quantifying:

$$\begin{aligned} &\forall y.\, \mathit{append}([], y, y) \leftarrow \mathit{true} \\ &\forall wxyz.\, \mathit{append}([w|x], y, [w|z]) \leftarrow \mathit{append}(x, y, z) \end{aligned}$$

To get the combined formula, we introduce fresh variables $v_1, \ldots, v_3$ and make the following definitions:

$$\begin{aligned} \psi' &\equiv \mathit{append}(v_1, v_2, v_3) \\ \phi'_1 &\equiv \exists y.\, v_1 = [] \wedge v_2 = y \wedge v_3 = y \\ \phi'_2 &\equiv \exists wxyz.\, v_1 = [w|x] \wedge v_2 = y \wedge v_3 = [w|z] \wedge \mathit{append}(x, y, z) \end{aligned}$$

Note that we have omitted *true* from the conjunctions, since they do not have any effect. Putting the definitions together we get the following clause completeness axiom:

$$\begin{aligned} &\forall v_1 v_2 v_3.\, \mathit{append}(v_1, v_2, v_3) \rightarrow \\ &\quad (\exists y.\, v_1 = [] \wedge v_2 = y \wedge v_3 = y) \vee \\ &\quad (\exists wxyz.\, v_1 = [w|x] \wedge v_2 = y \wedge v_3 = [w|y] \wedge \mathit{append}(x, y, z)) \end{aligned}$$

The completion of `append/3` is the same as the clause completeness axiom, except it uses a bi-implication:

$$\begin{aligned} &\forall v_1 v_2 v_3.\, \mathit{append}(v_1, v_2, v_3) \leftrightarrow \\ &\quad (\exists y.\, v_1 = [] \wedge v_2 = y \wedge v_3 = y) \vee \\ &\quad (\exists wxyz.\, v_1 = [w|x] \wedge v_2 = y \wedge v_3 = [w|z] \wedge \mathit{append}(x, y, z)) \end{aligned}$$

This is logically equivalent to the conjunction of the clause soundness and clause completeness axioms for `append/3`.

To illustrate the procedure for function definitions, we will of course look at the function `length/1`. We start with the clauses in their $\psi \leftarrow \phi$ form, as before:

$$\begin{aligned} &\mathit{length}([]) = 0 \leftarrow \mathit{true} \\ &\mathit{length}([x|y]) = 1 + \mathit{length}(y) \leftarrow \mathit{true} \end{aligned}$$

The clause soundness axioms are therefore:

$$\begin{aligned} &\mathit{length}([]) = 0 \leftarrow \mathit{true} \\ &\forall xy.\, \mathit{length}([x|y]) = 1 + \mathit{length}(y) \leftarrow \mathit{true} \end{aligned}$$

No quantifier is required on the first of these, since there are no free variables. To get the combined formula, we introduce fresh variables v_1 and v_r, and define:

$$\begin{aligned} \psi' &\equiv \mathit{length}(v_1) = v_r \\ \phi'_1 &\equiv v_1 = [] \wedge v_r = 0 \\ \phi'_2 &\equiv \exists xy.\, v_1 = [x|y] \wedge v_r = 1 + \mathit{length}(y) \end{aligned}$$

Putting these together we get the following clause completeness axiom:

$$\begin{aligned} &\forall v_1 v_r.\, \mathit{length}(v_1) = v_r \rightarrow \\ &\quad (v_1 = [] \wedge v_r = 0) \vee \\ &\quad (\exists xy.\, v_1 = [x|y] \wedge v_r = 1 + \mathit{length}(y)) \end{aligned}$$

As before, the function completion is the same as the clause completeness axiom with the implication replaced by a bi-implication.

3.7.7 Mode-determinism assertions

Modes and determinisms play a significant role in helping people understand Mercury code. They also play a part in the declarative semantics. For modes that are `det` or `multi` we generate an axiom that says that for every value of each of the inputs, there exists a solution. For modes that are `det` or `semidet` we generate an axiom that says that for every value of each of the inputs, there is at most one solution.

For example, consider the following modes for `append/3`:

```
:- mode append(in, out, in) is semidet.
:- mode append(out, out, in) is multi.
:- mode append(in, in, out) is det.
```

These three modes will generate the three axioms shown in Figure 3.7, respectively. Note that the third axiom, for the `det` mode, is the conjunction of the axioms that would apply for the `semidet` and `multi` modes.

$$\forall x y_1 y_2 z.\, append(x, y_1, z) \wedge append(x, y_2, z) \rightarrow y_1 = y_2$$
$$\forall z. \exists x y.\, append(x, y, z)$$
$$\forall x y. (\exists z.\, append(x, y, z)) \wedge$$
$$\quad (\forall z_1 z_2.\, append(x, y, z_1) \wedge append(x, y, z_2) \rightarrow z_1 = z_2)$$

Figure 3.7: Mode-determinism assertions for three modes of `append/3`.

One consequence of these axioms is that if a predicate has a mode where all arguments are inputs and where the determinism says it cannot fail, then any call to that predicate is logically equivalent to *true*. Unless a strict operational semantics is used, be aware that in most cases the compiler will optimize away such calls. If that happens then exceptions may not be thrown, trace goals may not be run, etc.

Similar axioms are generated for function modes that cannot fail. The axiom generated for a function's default mode states that the function is total—a vacuous statement in classical logic since function symbols are always interpreted as total functions. On the other hand, this presents a problem for functions that are `semidet` in the forward mode, that is, the mode with all arguments as inputs and the return value as an output. These denote *partial* functions, which cannot be directly represented in classical logic. We will discuss in Section 6.2 how these function modes are handled.

3.8 Classical semantics

3.8.1 Universes

A *universe* U is a non-empty set of *values* representing the domain of discourse.[3] The idea is that terms in the syntax correspond to values in U. If a term t corresponds to a value u in a given interpretation, we say that t denotes u, and that u is the denotation of t. For example, the code in Section 2.6 implementing arbitrary precision integers as lists of digits could have the set of all integers as its universe. In this interpretation, each list of digits would denote an integer.

The examples in Section 2.1 made use of Herbrand interpretations. In these the universe is the *Herbrand universe*, which is defined as the set of all ground data terms, and each ground data term simply denotes itself. For first order code the Herbrand universe can be considered as good as any other, since, given an interpretation in any other universe U, there exists a unique map from the Herbrand universe to U that ultimately leads to the same result.

[3]The term "domain" is also sometimes used instead of universe, but this is not quite the same as the domains that sometimes appear in the denotational semantics of other languages—there is no $\perp$ element, for example—so we will avoid using this term.

In the following discussion, for a universe U we will make use of the sets U^n for $n \geqslant 0$, where U^n is defined as the set of n-tuples of elements in U. These sets represent the possible argument values for predicates and functions of arity n.

3.8.2 Assignments

An *assignment* over a universe U is a mapping from variables to values in U. In the following we will use σ and ρ to stand for arbitrary assignments. For a variable x, the value that it maps to under σ is written as $\sigma(x)$.

If σ_1 and σ_2 are assignments such that $\sigma_1(v) = \sigma_2(v)$ for all variables v other than x, then we say that σ_1 differs from σ_2 only at x. Note that it can be the case that $\sigma_1(x)$ does still equal $\sigma_2(x)$. We write $\sigma\{v_1 \mapsto u_1, v_2 \mapsto u_2, \dots\}$, or just $\sigma\{v_i \mapsto u_i\}$, for the assignment that differs from σ only at each of the v_i, where it maps to u_i.

It's possible to imagine applying an assignment to formula, such that all of the free variables in the formula are replaced by the values they take in the assignment. If such an assignment makes the formula true, then the assignment is thought of as a *solution*. In the next two sections we will make this concept precise.

3.8.3 Interpretations

We have used the term "interpretation" a number of times to mean, intuitively, the way in which we understand the symbols appearing in a formula. For example, we have said that the symbol '+' can be interpreted as integer addition. The general idea of an interpretation is that it maps from syntactic elements—the predicate and function symbols—to some semantic universe.

Formally, an interpretation I over a universe U is a mapping defined on predicate and function symbols, such that:

- If f/n is a function symbol, then $I(f/n)$ is a total function $U^n \to U$. In other words, $I(f/n)$ takes n arguments from U and returns a value from U. For a constant a, $I(a)$ is just an element of U.

- If p/n is a predicate symbol, then $I(p/n)$ is a predicate (that is, a relation) over U^n. That is, $I(p/n)$ takes n arguments from U and returns either *true* or *false*.

Thus, the function $I(f/n)$ is the denotation of the function symbol f/n, and the predicate $I(p/n)$ is the denotation of the predicate symbol p/n.

More generally, we will need to show how an interpretation as defined above can be extended to a mapping on terms and formulas. Since these may contain variables, the result will depend on how values are assigned to those variables.

Given an interpretation I and an assignment σ, we can extend I to a mapping I_σ from terms to elements of U by applying the following rules:

- If x is a variable, then $I_\sigma(x)$ maps to $\sigma(x)$.

- If f/n is a function symbol and $t_1, \ldots, t_n$ are terms, then $I_\sigma(f(t_1, \ldots, t_n))$ maps to the function $I(f/n)$ applied to arguments $I_\sigma(t_1), \ldots, I_\sigma(t_n)$. For a constant a, $I_\sigma(a)$ just equals $I(a)$.

Similarly, this increasingly overloaded mapping can be extended to atomic formulas. The following rules are applied:

- $I_\sigma(\mathit{true})$ always maps to *true*.
- $I_\sigma(\mathit{false})$ always maps to *false*.
- If t_1 and t_2 are terms, then $I_\sigma(t_1 = t_2)$ maps to *true* if $I_\sigma(t_1)$ and $I_\sigma(t_2)$ map to the same value in U. Otherwise it maps to *false*.
- If p/n is a predicate symbol and $t_1, \ldots, t_n$ are terms, then $I_\sigma(p(t_1, \ldots, t_n))$ maps to the predicate $I(p/n)$ applied to arguments $I_\sigma(t_1), \ldots, I_\sigma(t_n)$.

Continuing, the mapping can be extended to compound formulas by applying the following rules:

- If ϕ is a formula constructed using a logical connective, then $I_\sigma(\phi)$ maps to a truth value via the usual (classical) truth table for that connective, using the truth values of I_σ for each sub-formula.
- If ϕ is of the form $\exists x.\, \psi$, then $I_\sigma(\phi)$ maps to *true* if $I_\rho(\psi)$ is true for some assignment ρ that differs from σ only at x. Otherwise it maps to *false*.
- If ϕ is of the form $\forall x.\, \psi$, then $I_\sigma(\phi)$ maps to *true* if $I_\rho(\psi)$ is true for all assignments ρ that differ from σ only at x. Otherwise it maps to *false*.

Finally, observe that if ϕ is a closed formula then $I_\sigma(\phi)$ is always the same regardless of the assignment. We can therefore define $I(\phi)$, without ambiguity, as being equal to $I_\sigma(\phi)$ where σ is an arbitrary assignment.

For example, consider an interpretation I in which $\mathit{append}/3$ is the list append predicate and $\mathit{length}/1$ is the list length function. Let σ be an assignment such that $\sigma(z) = [1, 2, 3]$, and let ϕ be the following formula.

$$\exists x y.\, \mathit{append}(x, y, z) \wedge \mathit{length}(x) = 2$$

We can evaluate $I_\sigma(\phi)$ as follows. Let $\rho = \sigma\{x \mapsto [1, 2], y \mapsto [3]\}$. That is, ρ is the assignment that differs from σ only at x and y, where it takes the values $[1, 2]$ and $[3]$, respectively. We have that $I_\rho(x)$ maps to $[1, 2]$, therefore $I_\rho(\mathit{length}(x))$ is the length of the list $[1, 2]$, which is 2. Thus $I_\rho(\mathit{length}(x) = 2)$ maps to *true*.

Similarly, $I_\rho(\mathit{append}(x, y, z))$ maps to *true*, since the arguments map to $[1, 2]$, $[3]$ and $[1, 2, 3]$, respectively, and the last of these is the result of appending the first two. Given this result and the result from the previous paragraph, we can see that $I_\rho(\mathit{append}(x, y, z) \wedge \mathit{length}(x) = 2)$ maps to *true*, via the truth table for '$\wedge$'.

Finally, $I_\sigma(\phi)$ maps to *true* because ρ is an assignment that differs from σ only at x and y, which satisfies our rule for the existential quantifier.

Had we used an assignment σ' with $\sigma'(z) = [1]$, then the append operation would have been false under the assignment $\rho' = \sigma'\{x \mapsto [1,2], y \mapsto [3]\}$, since $[1,2]$ and $[3]$ do not append to form $[1]$. Furthermore, no such assignment ρ' would be able to make the interpretation of the append operation true, and still have $I_{\rho'}(length(x))$ being true. As such, $I_{\sigma'}(\phi)$ would have evaluated to *false*.

3.8.4 Models

An interpretation can be thought of as one individual programmer's understanding of how a program works. If the programmer has an accurate picture in their head of the observable behaviour—the inputs and outputs—then their interpretation can be said to be a *model*.

Formally, let I be an interpretation over the universe U. We say that I is a model of a set of axioms if each of the axioms evaluates to true in that interpretation (recall that axioms are closed formulas, so we do not need to specify an assignment). If I is a model of the axioms generated by a program, then we say that I is a model of the program. Customarily, the variable M is used rather than I when discussing an interpretation that is a model.

We are now in a position to give the following.

Definition 2 (Declarative semantics)**.** *The declarative semantics of a Mercury program is the collection of models of that program.*

In other words, the declarative semantics is essentially all the possible ways of thinking about the program which accurately reflect how the program behaves.

Considering once again the arbitrary precision integers example that we saw in Section 2.6, one programmer might interpret the digit lists as integers directly. Another programmer might interpret them as lists of integers that will produce the actual integers oncce a particular function is applied. As long as their individual interpretations as a whole accurately reflect the program, then both programmers stand in equally good positions from which to make valid arguments about the program. Our definition of the declarative semantics as a set of models reflects this fact.

If there exists any model at all, then there are an infinite number of possible models. However, we generally do not have to consider all models as it is possible to get the same results by considering only a particular set of interpretations. For first-order code, it is sufficient to only consider models that are Herbrand interpretations.

Even so, there can be multiple Herbrand interpretations that are models of a program. For example, consider the program `p :- p.` The completion of this program is $p \leftrightarrow p$, which is a tautology. This means that it is true in every interpretation, specifically, p could be assigned the value *true* or the value *false*, but in either case the axiom would hold so both of these interpretations are models.

More concerning is the program `p :- not p.` In this case the completion is $p \leftrightarrow \neg p$, which is a contradiction. This means that it is false in every interpretation, which means that *there are no models*. Effectively, the behaviour of the entire program is undefined, at least in the classical semantics. This can be regarded as a moot point, however, since execution of a program that calls such a predicate would not terminate. Nonetheless, it motivates the following definition.

Definition 3 (Consistency)**.** *A theory is consistent if it contains no contradictions. That is, there is no formula ϕ such that both ϕ and $\neg\phi$ are contained in the theory.*

If there is at least one model of a theory then there cannot be any contradictions, so by this definition the theory must be consistent.

In the remainder of this guide we will assume we are working with a program whose axioms we denote by Γ. We assume that there is at least one model of Γ, so the program is consistent, We will say "model" to mean a model of Γ. With that in mind we give the following definitions.

Definition 4 (Satisfaction)**.** *Let ϕ be a formula. If M is a model and σ is an assignment such that $M_\sigma(\phi)$ is true, we say that M* satisfies *ϕ* under *σ, and that ϕ is true in M under σ. If ϕ is closed, we additionally say that M satisfies ϕ, and that ϕ is true in M. If there exists any model that satisfies ϕ under some assignment, we say that ϕ is* satisfiable*.*

Definition 5 (Validity)**.** *Let ϕ be a formula and σ an assignment. We say that ϕ is* valid under *σ, written '$\Gamma, \sigma \models \phi$', if every model satisfies ϕ under σ. If this holds for every possible assignment, in particular if ϕ is closed, we additionally say that ϕ is valid, and that ϕ is a* logical consequence *of Γ. This is written as $\Gamma \models \phi$.*

Definition 6 (Unsatisfiability)**.** *Let ϕ be a formula and σ an assignment. We say that ϕ is* unsatisfiable under *σ if there is no model M that satisfies ϕ under σ. If this holds for every possible assignment, in particular if ϕ is closed, we additionally say that ϕ is unsatisfiable.*

Returning to the definition of solution that we gave earlier, we can say that an assignment σ is a solution to a formula ϕ if and only if $\Gamma, \sigma \models \phi$. That is, σ is a solution for ϕ if every model satisfies ϕ under σ.

With the above definitions, no formula can be both valid and unsatisfiable. Furthermore, if a formula is valid then its negation is unsatisfiable, and vice-versa. Not all formulas are one or the other, however: if a formula is true in some models and false in others, then it is neither valid nor unsatisfiable. For example, consider again the program `p :- p.` Because there is a model in which p is true and another model in which p is false, the formula p is satisfiable but not valid. A formula like this that is neither valid nor unsatisfiable is said to be *contingent*.

A point about the $\models$ notation is worth underlining. We have used this symbol to denote the validity relation between sets of axioms, Γ, and closed formulas, ϕ, and the "valid under" relation between sets of axioms, assignments, and open formulas.

$$\forall y.\, append([], y, y) \leftarrow true$$
$$\forall wxyz.\, append([w|x], y, [w|z]) \leftarrow append(x, y, z)$$

Figure 3.8: Clause soundness axioms for *append*/3.

This symbol is commonly overloaded, in other ways, too. Some authors use it to denote the satisfaction relation between *models* and closed formulas, or between models, assignments, and open formulas. Other authors use it to denote the modelling relationship between interpretations and sets of axioms. That is, statements in the following forms may appear:

$$M \models \phi \qquad M, \sigma \models \phi \qquad M \models \Gamma$$

These mean, respectively, that M satisfies ϕ, that M satisfies ϕ under σ, and that M is a model of the set of axioms, Γ.

We will only need to use the forms given in our definition of validity, but the reader should be aware that the other forms may be used in other sources.

3.9 Example

Now that we have defined our semantics and specified what axioms generate a program's theory, how do we actually come up with a theorem? Theorems require proofs, so in this section we will give a couple of ad hoc arguments to try to prove that a formula is true.

Consider the formula $append([1], [2], [1, 2])$. We already know it is true given that *append*/3 is interpreted as list concatenation, but assume that we only know how it is defined, and not its interpretation. Can we prove it is true using just the axioms that are generated?

For convenience, the clause soundness axioms for *append*/3 from Section 3.7.6 are repeated in Figure 3.8.

First attempt

One approach to proving our formula is to try to determine directly which ground atoms are contained in some arbitrary Herbrand model of the program, which we will call H. If the formula we are interested in is contained in the model, then it must be true.

Looking at the axiom for the first clause, we can see that if we assign the value [2] to y, then the formula becomes:

$$append([], [2], [2]) \leftarrow true$$

From this we immediately get that $append([], [2], [2]) \in H$. We are allowed to choose, as we did, any value we like for y, since y is universally quantified—the axiom is applicable to all possible choices.

Now consider the axiom for the second clause. If we assign the values [], [2] and [2] to the variables x, y and z, respectively, then the right hand side becomes $append([], [2], [2])$, which we just established is in H. If we then assign to w the value 1, the formula becomes:

$$append([1], [2], [1, 2]) \leftarrow append([], [2], [2])$$

Thus we get that $append([1], [2], [1, 2]) \in H$, which completes our proof.

Our attempt at proving a formula has been successful, but there are a lot of ways to go about building a proof, and the proof we have arrived at has a drawback. It starts by proving a small fact via the first clause, then builds a larger fact via the second clause. Such a proof is known as a "bottom-up" proof. This is an interesting way of reasoning, and in fact it reflects, to an extent, how deductive databases go about answering queries. but it does not reflect how Mercury programs are executed.

Mercury execution starts with a high level goal, and reduces it down to smaller and smaller parts until each part can be solved directly using a fact clause. This style of proof is known as "top-down", in contrast to the bottom-up proof above. We will make a second attempt at proving our formula, this time using a top-down approach. While this will not completely describe how execution proceeds, it should help provide some intuition, and serve to motivate the operational semantics that will be the subject of the next chapter.

Second attempt

For our second attempt we will try a different strategy. We will start by assuming that the formula we wish to prove, $append([1], [2], [1, 2])$, is false. From that we will try derive a contradiction, which would show our assumption to be false, and thus prove that the formula is true. In other words, this will be a proof-by-contradiction. We refer again to the axioms in Figure 3.8.

Our approach will be to choose one of the axioms and try to match it to our formula by setting the arguments appropriately. Since the axioms are reverse implications for which we have assumed the left-hand side is false, we can infer that the right-hand side must also be false. By performing this inference a number of times in sequence we hope to reach a contradiction, that is, where the right-hand side is in fact true. We might fail to find such a contradiction, however. If we instead reach a tautology, it means we have failed to find the required contradiction. If we want to keep searching, we will need to go back to an earlier choice of axiom that we made, and choose the other axiom instead.

The predicate call we are interested in has argument values [1], [2] and [1, 2]. If we choose the axiom for the first clause, we find that there is no way to assign a

value to y such we can match the argument values, because the first argument will always be [].

We therefore choose the axiom for the second clause. By assigning the values 1, [], [2] and [2] to w, x, y and z, respectively, we get:

$$append([1], [2], [1, 2]) \leftarrow append([], [2], [2])$$

We have assumed the formula on the left-side side to be false, so the formula on the right-hand side must also be false.

Trying the first axiom on this new formula, we can assign the value [2] to y to obtain:

$$append([], [2], [2]) \leftarrow true$$

Again, the left-hand side is false so the right-hand side must also be false, but this is a contradiction because the right-hand side is *true*. We can therefore conclude that our original assumption must have been false. Thus $append([1], [2], [1, 2]) \in \boldsymbol{H}$, which completes our proof.

This style of proof more closely reflects how programs are executed. It is still missing one important thing, however, which is that there were no variables in the formula we proved. Programs in general have output variables which become bound in the course of execution, so our proof technique would need to take account of that. We also want to be able to perform the inferences in a way that is less ad hoc. That, ultimately, is the aim of the operational semantics.

We will cover the operational semantics in detail in the next chapter. Before that, however, we will make some brief philosophical remarks regarding classical logic.

3.10 Philosophical remarks

The semantics we have presented is classical. It is possible this will leave the reader with the impression that Mercury is for classicists and Mercury programmers must therefore believe that classical logic is the One True Logic. This impression would not be accurate, as a lot of consideration has gone into understanding the benefits and drawbacks of classical logic, and likewise in understanding what other logics have to offer.

The motivation for the focus on classical logic is straightforward: it provides an excellent trade-off between ease of reasoning and ability to observe a broad class of bugs. Its use does not, of course, preclude the addtional use of non-classical logic. In Chapter 7 we give a non-classical meaning to the logical connectives that enables more realistic reasoning about programs and their correctness with respect to a specification. As such, it demonstrates the usefulness of "logical pluralism" as a philosophy for reasoning about computer programs.

The underlying purpose of logic is to characterize how we think, not to tell us how to think. We humans have the innate ability to judge between a good argument

and a bad argument, and the best logic in a given situation is the one that most accurately reflects the way in which we exercise this ability.

Chapter 4

Operational semantics

4.1 Overview

In Chapter 3 we presented the declarative semantics of Mercury, and showed how it is possible to use the semantics to prove theorems about the program, and in particular about the solutions to formulas. The deductive system we used to construct these proofs, although its rules of inference were only hinted at informally, was essentially the standard one used with the predicate calculus.

The operational semantics of Mercury is a deductive system that, like the standard one, can be used to generate theorems. Unlike the standard one, it is based around a single rule of inference known as *SLD resolution*. This rule is able to give a top-down computational interpretation to a program, by which we mean that it defines the sequence of steps by which computation proceeds.

Generally, computation starts with a goal known as the *query*, and produces zero or more answers in the form of *substitutions*. We start this chapter by describing queries, substitutions, and unification, which are the key building blocks of the operational semantics.

We then introduce the SLD resolution inference rule, and show how *SLD trees* are defined. These give the operational semantics of "definite" logic programs, which are programs in which each clause body is a conjunction of atoms (that is, not using disjunction, negation, or if-then-else).

Some important results in the meta-theory, known as soundness and completeness, will then be covered. These meta-theorems demonstrate that the declarative semantics and the operational semantics give non-conflicting views of the program behaviour.

After this we extend our semantics to deal with Mercury goals in general. We introduce the negation-as-failure rule used in SLDNF resolution, which defines the behaviour of if-then-elses and negated goals. We provide a set of structural rules to deal with other Mercury goals.

Finally, we will briefly explain the origin of the phrase "SLD resolution", which may be something that has attracted the reader's curiosity.

4.2 Queries

Queries are (usually non-ground) goals that represent the starting point of a computation. Executing the query involves finding substitutions, which we refer to as *answers*, for which each ground instance corresponds to an assignment that makes the goal valid. A query essentially asks, "What are the assignments of the free variables for which this goal is valid?"

The initial query for the execution of a Mercury program is always a single call to `main/2`. It will, however, also be useful to consider queries that represent subcomputations within the program. For definite logic programs, such queries can be written in the following form:

```
:- Goal1, Goal2, ..., GoalN.
```

where each `GoalI` is an atomic goal, and commas are read as conjunction. If the conjunction of goals is in solved form, as defined in the next section, then no further computation is required: the corresponding substitution is the only answer.

The query is interpreted as the formula:[1]

$$\forall \bar{x}. \mathit{false} \leftarrow \phi_1 \wedge \ldots \wedge \phi_n$$

where $\phi_1, \ldots, \phi_n$ are the formulas corresponding to the goals, and $\bar{x}$ is the set of free variables occurring in the goals. Effectively, the query is interpreted as the *negation* of the goal, and the aim is to find all substitutions that make this negation false. Hence this is an attempt at proof by contradiction, similar to our proof from Section 3.9. That is, a proof, if found, is a refutation of the query, which in turn implies that the substitution is an answer to the goal.

Proof by contradiction may seem a roundabout way of doing things, and indeed some authors define the execution procedure directly to avoid this, but we define things this way because the resulting proof steps, when written out, have the premise on top and the conclusion underneath. This is the conventional way of writing proofs, but the choice ultimately makes no difference to the outcome.

4.3 Substitutions

A *substitution* is a partial mapping from variables to data terms, such that no variable that maps to a term occurs in any of the terms in the mapping. That is, variables on the left-hand side do not occur on the right-hand side. We write substitutions in the form:

```
{V1 = t1, ..., VN = tN}
```

[1] The origin of this notation, as with many other things, comes from theorem provers. The list of goals on the right-hand side of '`:-`' is taken as a conjunction, as we have done. The list of goals of the left-hand side is taken as a disjunction, and since in our case the list is empty, the disjunction is equivalent to *false*. As usual, free variables are implicitly universally quantified.

where `V1` to `VN` are variables, and `t1` to `tN` are arbitrary data terms (possibly variables themselves) that the variables respectively map to.

A substitution applied to an expression `t` yields an expression which is the same as `t`, but with each occurrence of a variable that maps to a term in the substitution replaced with the mapped term. A substitution can be applied to a goal in a similar way, by replacing each free occurrence of a variable with the term it maps to, if any.

Observe that a substitution without the braces is just a goal consisting of a conjunction of unifications. Indeed, substitutions can be thought of as goals that are in "solved form", in that applying them once to an expression or goal is straightforward and is sufficient to produce the entire effect (that is, substitutions are idempotent). The aim of computation is essentially to put goals into their solved forms.

Substitutions represent the state of computation: they record all we know about the variables so far. In Mercury, the instantiatedness of a set of variables describes the possible substitutions at that point in the code, which tells us something about what form the substitution must take. If the `inst` of a variable is `free` then the variable is not mapped to anything. If it is `bound` then the variable is mapped to a term whose principal functor is one of the ones listed, and whose arguments are described by the corresponding argument `insts`. If it is `ground` then the variable is mapped to a term that contains no variables.

While substitutions bear a resemblance to the assignments that we defined in Section 3.8.2, note that assignments map variables to elements of the universe. That is, assignments are semantic in nature, whereas substitutions map variables to data terms, and are thus syntactic.

4.4 Unification

Given two possibly non-ground data terms, unification is the process of finding a substitution on variables such that applying it to either term yields the same result. A substitution that makes two terms identical in this way is called a *unifier* of those terms.

Terms do not always have a unifier, in which case we say that the terms do not unify. If they do unify, however, there is always a "most general unifier" that does the least amount of binding possible. There may be more than one most general unifier, but they will be unique up to renaming of variables. The unification algorithm aims to find one such unifier.

For example, consider the terms `f(X,g(X))` and `f(h(Y),Z)`, and the substitution:

```
{X = h(Y), Z = g(h(Y))}
```

Applying this substitution to either of the terms yields the same result, namely `f(h(Y),g(h(Y)))`, so this substitution is a unifier. It is not difficult to see that it is a most general unifier.

The unification algorithm can be seen as a procedure for putting equations into solved form, that is, in the form of a substitution that is a most general unifier. Taking the above example, if the query is:

```
:- f(X,g(X)) = f(h(Y),Z).
```

then in solved form this would be:

```
:- X = h(Y), Z = g(h(Y)).
```

which corresponds to the substitution we had above.

In general, consider a query that is a set of equations as follows, where `s1` to `sN`, and `t1` to `tN`, are arbitrary data terms (possibly variables):

```
:- s1 = t1, s2 = t2, ..., sN = tN.
```

Initially, all of the goals are marked as unsolved. We first select a goal `G` that is not marked as solved. If there is no such goal then the algorithm terminates. We then apply one of the following rules, depending on the form `G` takes.

- If `G` is `X = X` for some variable `X`, remove it.
- If `G` is `f(s1, ..., sN) = f(t1, ..., tN)` for data constructor `f/N`, remove it and replace it with the set of equations `s1 = t1, ..., sN = tN`.
- If `G` is `f(s1, ..., sN) = g(t1, ..., tM)` for distinct data constructors `f/N` and `g/M`, the algorithm fails.
- If `G` is `t = X` for some non-variable data term `t` and variable `X`, remove it and replace it with `X = t`.
- If `G` is `X = t` where `t` is a data term not containing `X`, then replace all free occurrences of `X` elsewhere in the query by `t`. If `X` occurred freely in the original query then keep `G` and mark it as solved, otherwise discard it.
- If `G` is `X = t` where `t` is a non-variable data term and `X` occurs in `t`, the algorithm fails.

After applying the appropriate rule we go back and select another unsolved goal, or else terminate if there are none.

If the algorithm terminates without failing then the query will be in solved form, and the corresponding substitution will be a most general unifier of the equations. If the algorithm fails then the equations do not have a unifier. Note that the order in which goals are selected does not matter, since the results will be equivalent irrespective of the selection order.

A single unification can be solved as a special case of the above algorithm, by starting with the set containing just that equation. For our above example, three applications of the rules gives us the equation in solved form:

```
f(X,g(X)) = f(h(Y),Z)
          ⇓
  X = h(Y), g(X) = Z
          ⇓
  X = h(Y), Z = g(X)
          ⇓
X = h(Y), Z = g(h(Y))
```

Had this query included the goal `Y = Z`, the outcome would have been different, as we would eventually reach the equation `Z = g(h(Z))` and thus we would fail due to the last rule, which is the occurs check.

This algorithm described here is originally due to Martelli and Montanari. We can extend the algorithm to also allow for function calls in the goals, not just data terms as we currently do, however we will need to handle function calls differently in order to implement semantic rather than syntactic equality. We will see how to do that as part of the resolution algorithm, in the next section.

4.5 SLD Resolution

Deductive systems often use inference rules that follow a pattern of one introduction and one elimination rule for each logical symbol. This provides an elegant understanding of how the logic works, but for logic programming this view is not particularly useful since these rules do not provide any computational interpretation—they do not say how a program should be executed. Instead, at least for logic programs not using negation, there is one main inference rule. This rule is known as SLD resolution.[2]

Proofs start with a query in the form:

```
:- Goal1, ..., GoalN.
```

Each of the goals is atomic, meaning it is either a unification, a predicate call, or a logical constant. We assume for now that the program clauses are definite, so the body of each clause consists of a (possibly empty) conjunction of atomic goals. The more general case of Mercury clauses will be addressed in Section 4.9.

As discussed in Section 4.2, a query represents an assertion that the conjunction of goals is false for all variable assignments. The aim of SLD resolution is to derive a contradiction, thereby refuting the assertion. This occurs when the goals are in solved form, or there are no more goals left in the query. If such a contradiction is reached then the substitution corresponding to the solved goal is an answer.

The answer represents an assignment (or, if free variables remain, a set of assignments) for which the assertion is refuted, and therefore for which the conjunc-

[2]Historically, resolution was used as an inference technique in automated theorem provers. SLD resolution, which is an instance of this technique, was found to have a useful computational interpretation. It is from this that logic programming was developed.

tion of goals is valid. We refer to a derivation that results in a contradiction, and the substitution that it produces, as a *success*. We refer to the assignment or set of assignments that the answer represents as a *solution*.

Conversely, if at any stage the unification procedure fails then the derivation has reached a tautology. This means that we have failed to find a refutation, and we refer to this case as *failure*.

The third possibility is that neither a contradiction nor a tautology is found. We refer to this case as *nontermination*, but note that, aside from running forever, this also includes cases of abnormal termination such as throwing an exception.

The SLD resolution algorithm is parameterized by a selection function that returns a selected goal based on the current and previous queries. As before we will mark some goals as solved as we go, and require the selection function to choose a goal that is as yet unsolved. If the selected goal contains any function calls then the selection function also returns a selected function call from the goal.

The algorithm proceeds as follows. If there are no unsolved goals, the derivation succeeds. Otherwise, select an unsolved goal `G` using the selection function. Apply one of the following rules, depending on the form `G` takes.

- If `G` is `true`, delete it.
- If `G` is `false`, the derivation fails.
- If `G` is a unification between two data terms, handle it according to the rules given in the last section for the unification algorithm. If that fails then the derivation fails.
- If `G` is an atomic goal that contains a function call, and the selected function call in that goal is `f(t1,...,tN)` for a function `f/N`, then choose a clause whose head takes the form `f(s1,...,sN) = sR`. Rename variables as necessary so they do not conflict with any variables already present in the query. Remove the selected function call and replace it with `sR`, and to the query add the unifications `t1 = s1, ..., tN = sN`, followed by the clause body if present.
- If `G` is a predicate call `p(t1,...,tN)` for a predicate `p/N`, then choose a clause whose head takes the form `p(s1,...,sN)`. Rename variables as necessary so they do not conflict with any variables already present in the query. Remove the selected predicate call and replace it with the unifications `t1 = s1, ..., tN = sN`, followed by the clause body if present.

After applying the appropriate rule we go back and select another unsolved goal. If there are no such goals, the derivation succeeds.

The rules we have given are nondeterministic, in the sense that the order in which goals and clauses are selected is not fully specified. For the unification rules the order does not matter as the procedure will always lead to the same result, regardless of the choices made. The SLD resolution rules, on the other hand, require a bit more attention.

When it comes to selecting a goal, Mercury's selection function chooses a goal or function call whose initial `inst`s are satisfied by the argument terms. In the strict sequential semantics, the leftmost goal that satisfies this condition is selected. In the strict commutative semantics the selected goal need not be the leftmost one, but if there are any goals that were expanded from the body of a clause then one of the most recently expanded goals is selected.

For predicate and function calls, the algorithm also requires us to choose a clause from the program. In contrast to goal selection, where a single choice is committed to without going back and trying alternatives, clause selection results in a "choice point" being created. After exploring the derivations that follow from one choice, if more solutions are sought then execution *backtracks* to the most recent choice point and makes a different choice. If no such choice point exists, that is, if all choices have previously been explored, then execution of the query fails.

In Mercury, the `nondet` determinism category refers to the second form of non-determinism above, namely that relating to clause selection. Users do not need to declare the first form, as the compiler is responsible for making the choices in that regard.

The collection of possible derivations arising from a query can be arranged into a tree, where derivations branch off at each choice point in accordance with execution of the query. A tree of this form is known as an SLD tree.

To illustrate the algorithm, consider the call `append([a,b],[c],Xs)`. For convenience we repeat the clauses for `append/3` here:

```
append([], Bs, Bs).
append([V | As], Bs, [V | Cs]) :-
    append(As, Bs, Cs).
```

The initial query is as follows:

```
:- append([a,b], [c], Xs).
```

If the first clause were chosen we would get the unification `[a,b] = []`, among others, but this unification would fail. We therefore need to choose the second clause.

We rename this clause's variables by adding a numerical suffix, and replace the call with argument unifications and the renamed clause body. This results in the following query:

```
:- [a,b] = [V1 | As1], [c] = Bs1, Xs = [V1 | Cs1],
   append(As1, Bs1, Cs1).
```

After selecting each of the unifications and applying the unification rules, we get:

```
:- Xs = [a | Cs1], append([b], [c], Cs1).
```

If the first clause were chosen we would get the unification `[b] = []`, which would fail, so again we choose the second clause. We rename the clause variables with a different suffix and replace the call as before, which results in:

```
:- Xs = [a | Cs1], [b] = [V2 | As2], [c] = Bs2,
   Cs1 = [V2 | Cs2], append(As2, Bs2, Cs2).
```

Running unification rules again, we get:

```
:- Xs = [a,b | Cs2], append([], [c], Cs2).
```

This time we can choose the first clause, resulting in:

```
:- Xs = [a,b | Cs2], [] = [], [c] = Bs3, Cs2 = Bs3.
```

Running unification rules one last time we end up with:

```
:- Xs = [a,b,c].
```

which is the answer to the query.

Substitutions do not necessarily end up ground, as happened in this case, but in typical Mercury usage the modes will require that they do. In any case, for each ground instance of the answer, the derivation proves that the query formula is valid under the assignment corresponding to that ground instance. This motivates a definition to end the section.

Definition 7 (Provability)**.** *Let ϕ be a formula. We say that ϕ is* provable*, written $\Gamma \vdash \phi$, if there is some goal G with a successful derivation giving substitution S, such that ϕ is the formula corresponding to G with S applied to it. Furthermore, if $\Gamma \vdash \phi$ then $\Gamma \vdash \forall x.\, \phi$ for any variable x.*

4.6 Soundness and completeness

In the course of this chapter and the previous one a number of concepts have been introduced, some of which are essentially declarative in nature, others operational. In many cases the concepts come in pairs, one corresponding to the declarative view and the other to the operational view, which nonetheless reflect the same underlying concept.

Figure 4.1 shows some of the the correspondences between declarative concepts and their operational counterparts. Of particular importance is that between validity and provability, as that provides the basis for many of the other equivalences. The relationship between them is expressed by the following theorems.

Theorem 2 (Soundness)**.** *Let ϕ be any formula. If $\Gamma \vdash \phi$ then $\Gamma \models \phi$. That is, provability implies validity.*

Theorem 3 (Completeness)**.** *Let ϕ be any formula. If $\Gamma \models \phi$ then $\Gamma \vdash \phi$. That is, validity implies provability.*

Declarative concept		Operational concept
values	$\longleftrightarrow$	ground data terms
equality	$\longleftrightarrow$	unification
assignment	$\longleftrightarrow$	substitution
solution	$\longleftrightarrow$	answer
truth	$\longleftrightarrow$	success
falsity	$\longleftrightarrow$	failure
existence of model	$\longleftrightarrow$	consistency
validity	$\longleftrightarrow$	provability

Figure 4.1: Correspondences between declarative and operational concepts.

Between them, these theorems state that there is an equivalence between truth as expressed in the model, and truth as expressed by the program execution.

A deductive system like this, for which soundness and completeness holds, is sometimes referred to as a "full logic". In this context the declarative view is referred to as model-theoretic, while the operational view is referred to as proof-theoretic. The equivalence between the model-theoretic and proof-theoretic, as established by the theorems, can be expressed by the somewhat cute formula $\models \equiv \vdash$.

Of practical significance to programmers is that they can freely switch between thinking declaratively and thinking operationally. As we have seen, the former can allow for much simpler reasoning about programs than the latter. But, as we have also seen, this can only go as far as reasoning about partial correctness—there will always be situations where the programmer needs to reason operationally in order to verify correctness. Being able to freely switch between the two means programmers can use the declarative semantics most of the time, but can temporarily switch to the operational semantics when that is required.

It is, therefore, this correspondence between declarative and operational notions that justifies the use of the dual semantics of declarative programming, above and beyond the operational semantics that programming languages in general provide. In other words, we are justified in having two horizontal arrows in the lower part of Figure 1.1 on page 2, instead of one.

4.7 Operational incompleteness

At first glance, the Completeness theorem appears to put us in a kind of programmer's utopia. All that is required, it seems, is for the programmer to specify the logical outcomes they want, and the deductive system will know what to do.

Unfortunately, and perhaps unsurprisingly, this is not the case. A careful examination of the Completeness theorem shows that what the theorem states is that, if a formula is valid, there *exists* some proof that can be reached via application of the

```
p :- p.          q :- q, false.
p.
```

Figure 4.2: Two predicate definitions that illustrate issues with completeness.

resolution rule. The resolution rule, however, is not deterministic, and while execution nominally involves making all possible clause selection choices eventually, the compiler still has to commit to a particular clause ordering, as well as needing to commit to a particular goal selection at each stage. If the wrong choices are made, then a nonterminating derivation may be explored when there is in fact a successful or failed branch that would have been reached with different choices.

The code in Figure 4.2 illustrates this point. The predicate `p/0` has two clauses, the first of which immediately loops. If clause selection chooses this clause first for every call then the program loops indefinitely without succeeding. The second clause, however, proves that *p* is valid. As required by the Completeness theorem, the proof of this validity does exist—execution needs only to select the second clause at some stage—but with the above clause selection this proof will never be reached.

Similarly, the predicate `q/0` has a single clause whose body is comprised of two conjuncts. The first conjunct immediately loops, so if goal selection chooses this goal first for every call then the program loops indefinitely without failing. The second conjunct proves that *q* is unsatisfiable, and similarly to the previous example the proof would have been found if the second conjunct was selected at some stage.

For the purposes of programming, this situation is effectively a form of incompleteness, despite the theorem that says otherwise. This is why the Mercury Language Reference Manual talks about implementations being "at least as complete as" the strict commutative semantics. The term "complete" here refers to the form of effective completeness discussed in this section, rather than that discussed in Section 4.6.

The concept of completeness is used in many ways in mathematics. Indeed, in Section 2.5 we referred to clause completeness, which is another distinct usage of the word. It is thus helpful, in the context of logic programming, to explicitly refer to the effective form of completeness discussed in this section as *operational completeness*. The reader should be aware, however, that other authors commonly use the terms unqualified, which may cause confusion in some cases.

4.8 Negation-as-failure

So far we have been discussing definite logic programs, that is, programs without negation. This means programs without conditionals either, since they use a form of negation. In fact conditionals are more fundamental in Mercury, as '`not G`' is a shorthand for '`( if G then false else true )`'.

In the operational semantics, the rule for implementing negation is known as

negation-as-failure. The principle is easy to understand: if a goal succeeds then the negation of that goal should fail, and similarly, if a goal fails then the negation of that goal should succeed. We can implement negation-as-failure by adding an extra rule to the SLD rules from Section 4.5; such a system is known as SLDNF resolution.

Assuming that `G` is the selected goal, the additional rule for negation-as-failure is as follows:

- If `G` is a conditional goal of the form `( if GC then GT else GE )`, then execute `GC` as a new query. If the query succeeds with a substitution `S`, replace `G` with the result of applying `S` to `GT`. If the query fails without having produced a solution, that is, if all derivations have failed even after exhausting all possible choices, replace `G` with `GE`.

Note that if the condition succeeds, it may leave choice points behind. These will lead to alternative substitutions being applied to `GT`, leading to different derivations.

It is possible to extend the Soundness theorem to negation-as-failure if and only if the condition of the if-then-else does not cause any non-local variables to become instantiated. That is, resolving the condition should not lead to any variables that occur outside of the condition or then-branch to appear on the left hand side of an equation in the substitution. From the soundness of negation-as-failure, combined with our original Completeness theorem, We can also extend the Completeness theorem to programs with negation.

The requirement to ensure no non-local variables become instantiated can be challenging to verify manually. Fortunately, Mercury's mode system tracks changes to variable instantiation, so the compiler is able to perform the check at compile time without manual assistance.

4.9 Structural rules

We are now in a position to give rules that cover Mercury's other compound goals. Assuming that `G` is the selected goal, the additional rules are:

- If `G` is a disjunction, choose one of the disjuncts and replace `G` with the chosen disjunct. As with clause selection, a choice point is created so that execution may backtrack to the remaining disjuncts.

- If `G` is an explicit existential quantification, the quantified variables are renamed apart in the goal so that they do not conflict with any variables already in the query. The existentially quantified goal is then removed and replaced with the renamed goal.

- If `G` is defined as an abbreviation for another goal, it is removed and replaced with that other goal. Note that this rule is essentially applied at compile-time, since the replacement occurs as part of desugaring.

- If `G` is a purity or determinism cast, it is treated as if the cast was not present and executed in the same way as other goals.
- If `G` is a trace goal, the trace condition is evaluated (at compile-time or run-time, or both). If it is true then `G` is replaced by the goal with the trace condition, otherwise `G` is removed. If an I/O state is passed to the trace goal then the appropriate substitutions are made.
- If `G` is an event goal, it is removed. Doing so triggers a user-defined debug event that may be seen in `mdb`, the Mercury debugger.

4.10 What does SLD stand for?

The reader may be curious as to what the acronym "SLD" in SLD resolution actually means. Here is an explanation based on what we have covered in this chapter.

"S" stands for Selection function. The resolution procedure is parameterized by a selection function that dictates which goal to resolve next.

"L" stands for Linear. For each step in SLD resolution, a new query is generated from the previous one only, without needing to refer to other computations. As such, the proof tree consists of a single branch. This is referred to as a linear proof.

"D" stands for definite clauses. SLD resolution applies to logic programs consisting of definite clauses.

The full resolution rule is given the acronym SLDNF, where the "NF" stands for negation-as-failure. In the presence of negation the proofs are no longer linear, since they include sub-computations for negated goals. Also, obviously, the clauses are no longer definite. So perhaps SLDNF is a bit of a misnomer and only really makes sense in historical context, but nonetheless the name has stuck.

Chapter 5

The execution algorithm

5.1 Run-time unification

The abstract unification algorithm given in the previous chapter provides an overall view of the steps involved in solving unifications. Since Mercury code is compiled, however, many of these steps are able to be performed at compile-time and are thus not necessarily a major concern of the programmer.

In this section we describe the unification steps that are performed at run-time, which can be thought of as the residual steps left over after the compiler has done as much of the work as possible. This will allow programmers to get a better understanding of what kind of processor instructions will be needed, and how memory will be allocated and accessed.

The residual steps correspond to primitive unifications involving at most one function symbol, and which take the form `Y = X` or `Y = f(X1, ..., XN)`. These unifications are categorized further based on the results of mode analysis, which can infer either side of the equation as having mode `in`, mode `out`, mode `unused`, or some other mode.

Four categories of primitive unfications are compiled into inline code in the target language, which means they are executed with minimal overhead. The categories are as follows:

- Assignment unifications. These are instances of `Y = X` where one of the sides has mode `in` and the other has mode `out`. If `Y` is the output variable then we indicate such unifications as `Y := X`.

- Test unifications. These are instances of `Y = X` where both sides have mode `in`, and the type is a type constant (such as `int`, for example). We indicate such unifications as `Y == X`.

- Construction unifications. These are instances of `Y = f(X1, ..., XN)` where `Y` has mode `out` and each `Xi` has either mode `in` or mode `unused`. We indicate such unifications as `Y := f(X1, ..., XN)`.

```
append(As, Bs, Cs) :-
    As == [],
    Cs := Bs.
append(As, Bs, Cs) :-
    As == [X | As0],
    append(As0, Bs, Cs0),
    Cs := [X | Cs0].
```

Figure 5.1: The forwards mode of `append/3`, with unifications expanded and categorized as assignments, tests, constructions, and deconstructions.

- Deconstruction unifications. These are instances of `Y = f(X1, ..., XN)` where `Y` has mode `in` and each `Xi` has either mode `out` or mode `unused`. We indicate such unifications as `Y == f(X1, ..., XN)`.

Other instances of unification that are permitted by Mercury are compiled into calls to out-of-line predicates whose code is automatically generated by the compiler.

We can write a version of a predicate with unifications fully expanded (including head argument unifications). In a given mode of the predicate, we can indicate which category the unification is inferred to be in using the notation above.

Figure 5.1 shows how this would look for the forwards mode of `append/3`. The first clause performs a test on `As`, before assigning the value of `Bs` to `Cs`. The second clause deconstructs `As` into component arguments, makes a recursive call, then constructs `Cs` from the result.

5.2 Term representation

Generally speaking, a value in Mercury occupies a word, and possibly also an array of words allocated on the heap. Constants such as integer literals are stored in the word directly, whereas for terms built via a data constructor with arity > 0, the word contains a pointer to the heap array, which has one word for each argument.

(This is not the whole story. Some constructor arguments can be packed together more efficiently than indicated here, but to a first approximation this gives a reasonable indication of how much memory a term requires.)

Data constructors are represented by primary and/or secondary tag values, the former of which is stored in the pointer's unused low-bits and the latter of which is stored on the heap in an extra array element, or in the word's high-bits if it does not require a pointer.

The primitive unifications involve the following steps:

- For an assignment unification `Y := X`, we just need to copy the word from the location of `X` to the location of `Y`. If the word contains a pointer to a heap array, this array will be shared between both variables.

- For a test unification `Y == X`, we test the words for equality. If they are not equal, and if the words contain pointers to a heap array, we test that the tags are equal, and recursively test the arguments.
- For a construction unification `Y := f(X1, ..., XN)`, we allocate an array on the heap to hold the arguments and a secondary tag if required. We then fill in heap slots for each of the `Xi` with mode `in`, and the secondary tag if present. The word representing `Y` is the heap pointer, with a primary tag stored in the low-bits.
- For a deconstruction unification `Y == f(X1, ..., XN)`, we check that the tags for `f/N` are present, then use the pointer to dereference the heap slots for each `Xi` with mode `out`.

5.3 Switches

Chapter 6

Extensions

6.1 Higher-order code

In this section we show how higher-order code can be embedded in first-order logic. To do this, we need to define some additional abstract syntax, define a suitable universe, then provide a formal semantics. Here we only cover higher-order predicates; higher-order functions are handled in an analogous way.

Two additional pieces of abstract syntax are required: lambda terms which create higher-order values, and higher-order call formulas in which a higher-order value is applied to arguments. Lambda terms are written as follows:

$$\lambda v_1 \ldots v_n.\phi$$

This stands for the abstraction of ϕ over the variables $v_1, \ldots, v_n$, and corresponds to the Mercury expression:

```
(pred(V1::Mode1, ..., VN::ModeN) is Detism :- Goal)
```

If the arguments are not variables then fresh ones are introduced and unifications moved to the body, as is done with predicate completion. Higher-order calls are written as follows:

$$(t)(t_1, \ldots, t_n)$$

This stands for a call to the higher-order term t with $t_1, \ldots, t_n$ as arguments, and corresponds to the Mercury goal `(t)(t1, ..., tN)`.

In order to extend our first-order universe to handle higher-order terms, we need to add elements that the lambda terms denote. As with first-order predicates, lambda terms denote relations over the universe—that is, mappings from tuples to truth values—with the key difference being that these relations are also members of the universe itself.

For a first-order universe U, the set of all n-ary relations corresponds to the powerset of U^n. We might try to construct a higher-order universe by including the powerset along with the original set, then including all terms that can be constructed

from what we would then have, and so on. We would, however, inevitably end up with a set that supposedly includes its own powerset, but this would violate the theorem of Cantor that says this cannot happen. In a sense, the universe we have tried to define is too large to be considered a set. We can repair this situation by limiting our notion of powerset to only include *computable* relations.

A higher-order model constructed by limiting the powerset relation is known as a *general model*. If the powerset does not include every relation, then it cannot properly characterize the intended interpretation of second-order logic. However, for the purposes of programming language semantics, including just the computable relations ought to be sufficient to cover anything the programmer intends. A full semantics for second-order logic would be too powerful for our purposes.

With the universe we have just defined, we can give our semantics. Let t be the lambda term $\lambda v_1 \ldots v_n.\phi$ and let σ be an assignment. Given $u_1, \ldots, u_n \in U$, define σ' as $\sigma\{v_i \mapsto u_i\}$. Then $I_\sigma(t)$ is the relation such that $\langle u_1, \ldots, u_n \rangle$ maps to *true* in $I_\sigma(t)$ if and only if $I_{\sigma'}(\phi)$ is true.

Conversely, let t be any term denoting a higher-order value with arity n, and let ρ be an assignment. Given terms $t_1, \ldots, t_n$, we define $I_\rho((t)(t_1, \ldots, t_n))$ as true if and only if the tuple $\langle I_\rho(t_1), \ldots, I_\rho(t_n) \rangle$ maps to *true* in the relation $I_\rho(t)$.

We can express the above two logical equivalences more formally as follows:

$$\begin{array}{rcl} \langle u_1, \ldots, u_n \rangle \mapsto \mathit{true} & & \\ \quad \text{in } I_\sigma(\lambda v_1 \ldots v_n.\phi) & \iff & I_{\sigma'}(\phi) \text{ where } \sigma' = \sigma\{v_i \mapsto u_i\} \\ & & \\ I_\rho((t)(t_1, \ldots, t_n)) & \iff & \langle I_\rho(t_1), \ldots, I_\rho(t_n) \rangle \mapsto \mathit{true} \text{ in } I_\rho(t) \end{array}$$

Free variables in the lambda expression, that is, free variables in ϕ other than the v_i, are assigned values by σ, which is the assignment for the formula where the lambda term occurs. The v_i, on the other hand, are assigned values by ρ, which is the assignment for the formula where the higher-order call occurs.

Lambda terms are implemented with "closures", which are ground data terms that consist of a code pointer, along with a ground data term for each of the free variables in the lambda term. The ground data terms come from the substitution at the point where the lambda term is constructed—in our operational semantics, free variables in the lambda expression have substitutions applied to them by earlier unifications, in the same way that other free variables do.

The code pointer is to a piece of code generated by the compiler. The generated code represents a predicate whose arguments consist of the free variables in the lambda term, followed by the v_i variables. Implementing the higher-order call involves appending ground data terms for the higher-order call arguments to the ground data terms in the closure, then jumping to the code pointer with these ground terms as the arguments.

6.2 Partial functions

We mentioned in Section 3.6 that, while the predicate calculus requires that all functions be total, Mercury allows them to be partial in the form of `semidet` functions. Classically, every term denotes something, but for a `semidet` function applied to arguments outside the function's domain (that is, where the function application fails) nothing is denoted.

Such terms are sometimes called non-denoting terms, and a logic that allows non-denoting terms is called a free logic. We define the semantics by treating as false any atomic formula containing a non-denoting term; this approach is known as negative free logic.

In negative free logic, an "existence check" is required for each partial function called within an atomic formula, except those that are already of the form $y = f(t_1, \ldots, t_n)$. The existence check succeeds if and only if the partial function term does actually denote something.

The existence check can be implemented by equating the function call with a fresh variable, and replacing the function call in the atomic formula with the variable we have just introduced. The original atomic formula is then replaced with the conjunction of the new variable equation and the new atomic formula, with the new variable existentially quantified.

That is, if t is a partial function call occurring in the atomic formula ϕ, v is a fresh variable, and ϕ' is ϕ with the sub-term t replaced by v, then ϕ is replaced as follows:

$$\phi \qquad \text{becomes} \qquad \exists v.\, v = t \wedge \phi'$$

If in $v = t$ the call to t fails, then t is non-denoting, so the conjunction fails as negative free logic required the original atomic formula to do. Thus, since $\exists$ only quantifies over values in the universe, existentially quantifying the fresh variable quite literally performs the existence check.

For example, consider the following declaration for a function that returns the nth element of a list, or fails if n is out of range.

```
:- func index(list(T), int) = T is semidet.
```

The goal

```
p(index(L, 3))
```

would be translated into

```
some [V] ( V = index(L, 3), p(V) )
```

If `L` has fewer than three elements then `index(L, 3)` is non-denoting, and the call to `p` should therefore fail. And indeed the translated goal would, since there is no value for `V` for which the formula is true.

In our operational semantics from Section 4.5, function calls are handled in such a way as to be equivalent to the above. The clause return value is used directly

instead of introducing the variable `V`, but in the end the effect is the same as the translation we have just given, since renamed variables from the clause behave the same as existentially quantified variables.

Negative free logic has some features often considered undesirable. For example, the goal `index(L,N) = index(L,N)` is false if the index is out of range, even though syntactic identity suggests it ought to be true. We have defined equality semantically, however, in that two terms are equal if and only if they denote the same thing. As such, a non-denoting term can never be equal to another term, including itself.

Perhaps more concerning is the effect on substitutivity. Consider the following two goals:

```
index(L, N) \= a

V = index(L, N),
V \= a
```

Substitutivity would suggest that these are equivalent, but in fact if the index is out of range then the first succeeds but the second fails. The explanation is that `A \= B` is not an atomic formula, it is an abbreviation for `not (A = B)`. Any existence check for `A` or `B` needs to be put inside the negation, conjoined with the underlying atomic formula. The proper translation is thus:

```
not some [V] (V = index(L, N), V = a)
```

As a consequence of this, understanding the code requires knowing which goals are considered atomic, and which look atomic but are actually abbreviations.

If any of these effects leave an unpleasant taste, the best advice is to avoid `semidet` functions where possible and use `semidet` predicates instead. Functions are allowed to be `semidet` because such functions are the natural way to interpret field access functions where there is more than one constructor in the type. There is no obligation for users to write other functions in this way, however, and if there are such functions to be called, they can always be wrapped in a `semidet` predicate.

6.3 Exceptions

What is the declarative semantics of throwing an exception? This may seem an obvious question, since a declarative semantics is provided for catching exceptions. But despite this, at the time of writing the Mercury documentation does not give a clear answer.

It might be tempting to just say that, declaratively, throwing an exception is the same as being false. In both cases, no variable bindings are produced. This does not work out, however, since an exception thrown from inside a negation should be the same as an exception thrown from outside. If exceptions are meant to be false then negated exceptions must be true, which breaks our own rule.

The operational requirements are sufficiently understood: resolving an exception must neither succeed nor fail. There can be no problem with soundness, since success and failure are the only results that would constrain the models, but this is not the case for completeness. In order to maintain completeness, throwing an exception cannot be valid as that would require success, and it cannot be unsatisfiable as that would require failure.

There is a third option, however, which is that throwing an exception can be considered contingent. That is, there exists a model in which it is true and another in which it is false. With this arrangement the deductive system can arguably be considered complete, since it would not be required to prove anything at all in the case of thrown exceptions.

Another way of saying this is that, declaratively, the throw predicate may be defined as follows.

```
:- pred throw(T::in) is erroneous.
throw(X) :- throw(X).
```

It is thus declaratively equivalent to a loop. Operationally, of course, it immediately returns to the closest enclosing catch rather than running forever.

One thing to be aware of when programming with exceptions is that, in some cases, the mode-determinism assertions imply that a given call is equivalent to *true*. As mentioned in Section 3.7.7, unless a strict operational semantics is used the compiler may optimize away such calls (the default semantics is strict, so this does not happen unless a non-default semantics is explicitly selected). Thus, if the call is intended to throw an exception, the exception may not end up being thrown.

The same issue can arise with the semantics we give here. Consider the following goal:

```
( if throw(X) then true else true )
```

Our semantics says this goal is true in any given model, irrespective of whether or not *throw*(x) is true in that model. The compiler would therefore be justified in replacing this goal with *true*, meaning that the exception does not get thrown. As discussed above, however, it will not do so if the operational semantics is strict.

6.4 Types

So far we have largely ignored types, so the axioms we gave in Section 3.7 are technically incorrect. However, if we have unary predicates that correspond to each type, we can adjust the quantifiers to account for types. This process is known as *relativization*.

For example, given a type T and a variable x of this type, we can quantify the variable in the formula ϕ by writing $\forall x : T.\, \phi$ and $\exists x : T.\, \phi$. If the predicate corresponding to this type is p_T, then these formulas can be treated as abbreviations for $\forall x.\, p_T(x) \rightarrow \phi$ and $\exists x.\, p_T(x) \wedge \phi$, respectively. This will ensure that only well-typed terms will play a role in the semantics.

Chapter 7

Non-classical models

This section is not yet written. See the following for the main ideas:

NAISH, L., & SØNDERGAARD, H. (2014). Truth versus information in logic programming. *Theory and Practice of Logic Programming, 14(6)*, 803-840.

Appendix A

Glossary index

answer (pp. 42, 46) The substitution that results from a successful computation.

assignment (p. 33) A mapping from variables to values in the universe. (p. 53) A unification of the form `Y = X`, where one of the sides has mode `in` and the other has mode `out`.

atomic goal (p. 45) A unification, predicate call, or logical constant.

axiom (p. 25) A closed formula that is taken to be true without proof.

backtrack (p. 47) When executing a query, to jump to the most recent choice point, if one exists, in order to resume execution with a different choice.

bound (p. 21) A variable in a formula that is captured by a quantifier is said to be bound. (p. 43) A variable that is mapped by a substitution is said to be bound.

choice point (p. 47) A point in a computation at which a clause or disjunct selection was made, and which can be backtracked to in order to execute alternative branches.

clause completeness (p. 13) Of a predicate or function definition, having clauses that cover every possible input that has a solution.

clause soundness (p. 13) Of a clause defining a predicate or function, producing solutions that are true in the intended interpretation.

closed formula (p. 22) A formula in which there are no free variables.

closure (p. 58) The representation of a lambda term. It consists of a code pointer, along with values for the free variables in the lambda term.

completeness (pp. 48, 51) A deductive system is complete if validity implies provability. Completeness, in reference to a deductive system, is not to be confused with operational completeness or clause completeness.

completion (p. 30) Conversion of a set of clauses that define a predicate or function into a single closed formula.

consistent (p. 36) A theory is consistent if it does not contain any contradictions.

constant (p. 20) A function that does not take any arguments.

construction (p. 53) A unification of the form `Y = f(X1, ..., XN)`, where `Y` has mode `out` and each `Xi` has either mode `in` or mode `unused`.

contingent (pp. 36, 61) True in some model, and false in some other model.

data term (p. 25) A term that does not contain any function calls.

declarative debugging (pp. 14–15) A debugging algorithm based on comparing the declarative behaviour of the program with the intended interpretation.

declarative semantics (pp. 5, 35) The collection of models of a Mercury program.

deconstruction (p. 54) A unification of the form `Y = f(X1, ..., XN)`, where `Y` has mode `in` and each `Xi` has either mode `out` or mode `unused`.

definite (p. 41) A clause is definite if it is either a fact, or its body is a conjunction of atomic goals.

failure (p. 46) A derivation that terminates without producing an answer. (p. 46) Finishing execution of a query after exhausting all choice points.

free (p. 21) A variable in a formula that is not captured by a quantifier is said to be free. (p. 43) A variable that is not mapped by a substitution is said to be free.

full logic (p. 49) A logic equipped with a deductive system that is both sound and complete.

ground (p. 43) A variable that is mapped by a substitution to a term containing no variables is said to be ground. A term that contains no variables is also said to be ground.

Herbrand interpretation (p. 5) An interpretation where the universe of values is just the set of ground data terms.

Herbrand universe (p. 32) The set of all ground data terms.

intended interpretation (p. 7) The interpretation of a specification.

interpretation (pp. 7, 33) A mapping from syntactic elements to the semantic domain.

logical (p. 11) Of a predicate or function, having a consistent declarative semantics across all calls; pure. (p. 20) Pertaining to the symbols that are a fixed part of the language, in contrast to the predicate and function symbols defined by the program.

missing answer (p. 13) One of the two classes of bugs that are observable in the declarative semantics. An answer is missing if it is false according to the program as written, but true in the intended interpretation. Also see *wrong answer*.

model (p. 35) An interpretation under which a set of axioms are all true.

negation-as-failure (p. 50) An inference rule that says if a goal succeeds, then the negation of that goal should finitely fail, and vice versa.

negative free logic (p. 59) A logic that permits non-denoting terms, and treats atomic formulas containing non-denoting terms as being false.

non-denoting term (p. 59) A term that contains a call to a `semidet` function, with arguments for which the function fails.

non-logical (p. 11) Of a predicate or function, not having a consistent declarative semantics across all calls; impure. (p. 20) Pertaining to the predicate and function symbols defined by the program, in contrast to the symbols that are a fixed part of the language.

occurs check (p. 27) A check that a variable is not bound to a term that contains that variable. Also know as an occur check.

operational completeness (p. 49) The extent to which an operational semantics is able to avoid unnecessary nontermination.

operational semantics (pp. 41–49) The computation defined by a program.

partial correctness (p. 13) Correctness according to the declarative semantics. It does not consider issues such as computational complexity.

provable (p. 48) A closed formula for which there exists a successful derivation is said to be provable. Can be written as $\Gamma \vdash \phi$.

pure (p. 9) Of a predicate or function, having a consistent declarative interpretation across all calls, regardless of the mode in which the call is made.

query (p. 42) The initial goal for a computation.

relativization (p. 61) Adding explicit type checks that cause failure, to an otherwise untyped logic program.

resolution (p. 45) A type of inference rule. Logic programming uses SLD resolution as its primary mechanism of computation.

satisfiable (p. 36) A model satisfies a closed formula if it maps the formula to *true*. If a model exists that satisfies a formula, that formula is said to be satisfiable.

semantic equality (p. 24) The relation between terms that holds if and only if both terms denote the same value.

SLD tree (p. 47) A tree formed from SLD derivations by including choice point nodes, where the derivations branching off from that choice point are the child nodes.

solution (pp. 33, 36, 46) An assignment or set of assignments under which, in every model, a formula is true.

solved form (pp. 43, 44) A goal that takes the form of a substitution.

soundness (pp. 48, 51) A deductive system is sound if provability implies validity.

substitution (p. 42) A partial mapping from variables to terms, such that no variable that maps to a term occurs in any of the terms in the mapping.

success (p. 46) A derivation that produces an answer.

test (p. 53) A unification of the form `Y = X`, where both sides have mode `in`.

theorem (p. 25) A closed formula that is provable from a set of axioms. The collection of all such closed formulas is known as a theory.

unification (p. 43) The process of finding the most general substitution that makes two terms identical.

Unique Names Assumption (p. 25) The assumption that two ground data terms are equal only if they are syntactically identical.

unsatisfiable (p. 36) False in all models.

valid (p. 36) True in all models.

value (p. 32) An element of the semantic universe. Values are denoted by data terms.

wrong answer (p. 13) One of the two classes of bugs that are observable in the declarative semantics. An answer is wrong if it is true according to the program as written, but false in the intended interpretation. Also see *missing answer*.