In this module, we shall investigate proofs about complicated objects, such as computer programs. It has five sections.

**Equality.** A look at equality in predicate logic.

**Arithmetic.** A familiar setting, which we use to formalize inductive arguments.

**Lists and Programs.** Extending inductive arguments to binary structures, especially those structures that occur in computer programs.

**Proofs about Programs.** A deeper investigation, looking at entire programs as structures.

**What Programs—and Logic—Can’t Do.** Limits to the power of programs and logic.

## 1 The Equality Relation

Over any domain \( D \), one very useful relation is the equality relation. Formally, this relation is the set of pairs that have the same first and second element: \( \{ \langle x, x \rangle \mid x \in D \} \). The equality relation satisfies the following axioms.

**EQ1:** \( \forall x \ x = x \) is an axiom.

**EQ2:** For each formula \( \varphi \) and variable \( z \), \( \forall x \ \forall y \ (x = y \to (\varphi[x/z] \to \varphi[y/z])) \) is an axiom.

Axiom EQ1 states that everything is equal to itself. The schema EQ2 reflects that equal things have exactly the same properties. We can obtain further familiar properties of equality by inference.

1.1. **Lemma.** The EQ axioms imply that the relation “=” is symmetric and transitive.\(^2\) That is,

\[
\vdash \forall x \ \forall y \ (x = y \to y = x) \quad (\text{symmetry})
\]

and

\[
\vdash \forall w \ \forall x \ \forall y \ (x = y \to (y = w \to x = w)) \quad (\text{transitivity}).
\]

**Proof.** For the first, choose \( \varphi \) to be \( z = x \) in Axiom EQ2, yielding \( \forall x \ \forall y \ (x = y \to (x = x \to y = x)) \). Then EQ1 and modus ponens yield the required formula \( \forall x \ \forall y \ (x = y \to y = x) \).

---

\(^1\)Permission is granted to the University of Waterloo to use this material for teaching and research within the University.

\(^2\)Axiom EQ1 is reflexivity; thus equality is an equivalence relation, as expected.
Symmetry:

1. \( b = b \quad \text{EQ1} + \forall e \ [b \text{ fresh}] \)
2. \( a = b \rightarrow (b = b \rightarrow b = a) \quad \text{EQ2} + \forall e \ [z = x; a \text{ fresh}] \)
3. \( a = b \quad \text{Assumption} \)
4. \( b = b \rightarrow b = a \rightarrow e: 3, 2 \)
5. \( b = a \rightarrow e: 1, 4 \)
6. \( a = b \rightarrow b = a \rightarrow i: 3–5 \)
7. \( \forall x \forall y (x = y \rightarrow y = x) \quad 2(\forall i): 6 \)

Transitivity:

1. \( b = a \rightarrow (b = c \rightarrow a = c) \quad \text{EQ2} + \forall e \ [z = c; a, b, c \text{ fresh}] \)
2. \( a = b \quad \text{Assumption} \)
3. \( a = b \rightarrow b = a \quad \text{Symmetry of } = \)
4. \( b = a \rightarrow e: 2,3 \)
5. \( b = c \rightarrow a = c \rightarrow e: 4,1 \)
6. \( a = b \rightarrow (b = c \rightarrow a = c) \rightarrow i \)
7. \( \forall w \forall x \forall y (x = y \rightarrow (y = w \rightarrow x = w)) \quad \forall i (\times 3): 6 \)

Figure 1: Proofs of symmetry and transitivity of equality

For the second formula, let \( \varphi \) be \( z = w \) in Axiom EQ2, yielding \( \forall x \forall y (x = y \rightarrow (x = w \rightarrow y = w)) \).

Generalization of the free variable \( w \) then yields the required formula.

The full proofs\(^3\) appear in Figure 1.

2 Arithmetic

The natural numbers form a basic concept of mathematics. They derive from the fundamental notion of counting things. We have a number zero for no things.\(^4\) Once we have counted some things, if we find a next thing, we have a number to count it as well—different than all of the previous numbers. The set of all natural numbers is nothing more nor less than the collection of all numbers we can reach this way.\(^5\)

\(^3\)Full, that is, except for a few abbreviations:
- We do not copy axioms into the proof; we simply refer to them. In the case of \( \forall i \) applied to an axiom \( A \), we refer to this specialization as \( "A + \forall e" \). In addition, we may “collapse” a use of modus ponens with the specialization step. (The referenced line determines which formula was required as an axiom.)
- We shall not use a separate step to introduce fresh variables. We shall simply indicate them in the explanation for the step. Similarly, we shall combine \( \forall \)-introduction steps together as one.

\(^4\)Originally, people counted starting with one. But mathematicians eventually realized that zero is a very useful number to have, and thus it got included.

\(^5\)Thus, for example, more exotic things like 1/2 or \(-1\) are not natural numbers—we can't count up to them.
PA1: \( \forall x \, s(x) \neq 0. \)

PA2: \( \forall x \, \forall y \, (s(x) = s(y) \rightarrow x = y). \)

PA3: \( \forall x \, (x + 0 = x). \)

PA4: \( \forall x \, \forall y \, x + s(y) = s(x + y). \)

PA5: \( \forall x \, x \cdot 0 = 0. \)

PA6: \( \forall x \, \forall y \, x \cdot s(y) = x \cdot y + x. \)

PA7: For each variable \( x \) and formula \( \varphi(x) \),

\[ \varphi[0/x] \rightarrow ((\forall x \, (\varphi \rightarrow \varphi[s(x)/x])) \rightarrow \forall x \, \varphi) \]

Figure 2: Axioms for Peano Arithmetic

Using the symbol “0” for the initial number, and the symbol “s” for “the next number” (or “successor”), we have a term for every natural number:

\( 0, s(0), s(s(0)), s(s(s(0))), \ldots. \)

We augment this set of numbers by operations that have proven useful, such as addition, multiplication, etc.

The Peano axioms\(^6\) for arithmetic are given in Figure 2. Taken together with the equality axioms above, they yield most of the known facts about the natural numbers. Axioms PA1 and PA2 give the basic structure of the natural numbers. Starting from the constant 0, there a succession of natural numbers; different numbers have different successors; and 0 is not a successor. Axioms PA3 and PA4 give a recursive (inductive) characterization of addition. Axioms PA5 and PA6 do the same for multiplication. The schema of Axiom PA7 justifies the use of recursion: the natural numbers satisfy the induction principle.\(^7\)

These axioms imply all of the familiar properties of the natural numbers. For example, we can derive that addition is commutative.

\[ \text{2.1. Theorem. Addition in Peano Arithmetic is commutative; that is, there is a proof of the formula} \]

\[ \forall x \, \forall y \, x + y = y + x \]

\[ \text{from the axioms.} \]

How can we find such a proof? Let’s take it step by step. In a formal proof, the only available properties of “+” are those given by the axioms. Combining Axioms PA3 and PA4 with Axiom PA7 (induction) will yield the required result.

\(^6\)Named in honour of Giuseppe Peano, who first collected them together and studied their properties. The formulation here is not identical to Peano’s, but any of the equivalent axiom sets, and also similar systems of second-order logic, are called “the Peano axioms.”

\(^7\)The formula \( \varphi \) appearing in the schema represents the “property” to be proved.
We need to choose a formula \( \varphi \) to use in Axiom PA7. To make the conclusion of PA7 be the formula we need to prove, we can choose \( \varphi \) to be the formula \( \forall y \ x + y = y + x \). With this choice of \( \varphi \), we can get the required formula via modus ponens, provided that we can obtain the formulas \( \varphi(0) \) and \( \varphi \rightarrow \varphi[s(x)/x] \). Thus these formulas become goals of sub-proofs.

We start with the first goal, which is \( \forall y \ 0 + y = y + 0 \). Since PA3 gives the value of \( y + 0 \), namely \( y \), we try to show that \( 0 + y \) also has value \( y \)—that is, to derive the formula \( \forall y \ (0 + y = y) \). How to do so? The only method available is induction: in order to complete the proof by induction, we must do a proof by induction. To help keep it all straight, let’s pull out the “inner” statement and make a lemma out of it.

2.1.1. Lemma. Peano Arithmetic has a proof of the formula \( \forall y \ 0 + y = y \).

Proof of lemma. To obtain the target formula via Axiom PA7, we must find proofs of the formulas \( 0 + 0 = 0 \) and \( \forall y \ (0 + y = y \rightarrow 0 + s(y) = s(y)) \). The former is a specialization of PA3. For the latter, PA4 gives \( 0 + s(y) = s(0 + y); \) also EQ2 yields \( 0 + y = y \rightarrow s(0 + y) = s(y) \).

Since \( y + 0 = y \) (by PA3), we get \( s(y + 0) = s(y) \) from Axiom EQ2. PA3 also gives \( s(y) = s(y) + 0 \). Thus the hypothesis \( 0 + y \) and the axioms of equality yield \( 0 + s(y) = s(y) \), as we want.

Using PA7 and modus ponens completes the required proof of the lemma. The detailed proof:

1. \( 0 + 0 = 0 \) \( \quad \) PA3 + \( \forall e \)
2. \( y + 0 = y \) \( \quad \) PA3 + \( \forall e \) \( [y \ \text{fresh}] \)
3. \( s(y + 0) = s(y) \) \( \quad \) EQ2 + \( \forall e + \text{MP: 2} \)
4. \( 0 + s(y) = s(0 + y) \) \( \quad \) PA4 + \( \forall e \)
5. \( 0 + y = y \) \( \quad \) Assumption
6. \( s(0 + y) = s(y) \) \( \quad \) EQ2 + \( \forall e + \text{MP: 5} \)
7. \( 0 + s(y) = s(y) \) \( \quad \) =-trans: 4, 6
8. \( 0 + y = y \rightarrow 0 + s(y) = s(y) \) \( \quad \) \( \rightarrow \text{i: 5–7} \)
9. \( \forall y \ (0 + y = y \rightarrow 0 + s(y) = s(y)) \) \( \quad \) \( \forall i: 8 \)
10. \( \forall y \ y + 0 = y \) \( \quad \) PA7 + \( \forall e + \text{MP}^2: 1, 9 \)

For the second goal of the theorem, we use a second lemma.

2.1.2. Lemma. For each free variable \( x \),

\[
\forall y \ x + y = y + x \vdash_{PA} \forall y \ s(x) + y = s(x + y).
\]

Proof of lemma. We use induction on variable \( y \). The basis \( s(x) + 0 = s(x + 0) \) we have essentially already done. For the induction step, we require \( \forall z \ (s(x) + z = s(x + z) \rightarrow s(x) + s(z) = s(x + s(z))) \); the necessary equalities come from specializing the assumption both to \( z \) and to \( s(x) \). Then Axiom PA7 and modus ponens complete the proof.

In detail,
### Proof of Theorem 2.1. The proof of commutativity is a combination of the proofs in the two lemmas and a few lines to connect them and achieve the final result. Overall, the full formal proof looks as follows.

1. \(0 + 0 = 0\) \(\text{PA3} + \forall e\)
2. \(\ldots [\text{proof from Lemma 2.1.1}]\)
3. \(\forall y \, (0 + y = y \rightarrow 0 + s(y) = s(y))\) \(\text{PA7} + \forall e + \text{MP}^2: 1, 12\)
4. \(\forall y \, 0 + y = y\) \(\text{PA3} + \forall e + =\text{-trans}\)
5. \(\forall y \, x + y = y + x\) \(\text{Assumption}\)
6. \(\ldots [\text{proof from Lemma 2.1.2}]\)
7. \(\forall y \, x + y = y + x \rightarrow \forall y \, s(x) + y = s(y) + x\) \(\rightarrow i: 15–26\)
8. \(\forall x \, \forall y \, x + y = y + x\) \(\text{PA7} + \forall e + \text{MP}^2: 12, 27\)

This completes the proof of Theorem 2.1. \(\square\)

The other familiar properties of addition, and of multiplication, have similar proofs. One can continue and define divisibility, primeness, and many other properties.

### 2.2. Exercise. We can express the statement, “Every non-zero natural number has a predecessor” by the formula

\[\forall x \, (x \neq 0 \rightarrow \exists y \, s(y) = x)\]  

Show that this formula has a proof from the PA axioms.

### 2.3. Exercise. Write formulas that express each of the following properties.
1. (a) $x$ is a composite number.
    (b) $x$ is a prime number.

2. If $x$ divides $y$ and $y$ divides $z$, then $x$ divides $z$.

2.4. Exercise.

1. Prove formally that addition is associative: $x + (y + z) = (x + y) + z$. (Note: this is rather simpler than the proof for commutativity, above. Don’t let it scare you. The choice of variable for the induction does matter: if you get stuck one way, try another. [Or you can make use of commutativity.])

2. Prove that multiplication distributes over addition on the left; that is, $(x + y)z = xz + yz$.

3. Prove that multiplication distributes over addition on the right; that is, $x(y + z) = xy + xz$. Do not assume commutativity of multiplication. (If your induction seems to require commutativity of multiplication, try a different induction.)

4. Prove that multiplication is associative and commutative. Follow the corresponding proofs for addition.
3 Lists, and Programs

One can, if desired, use the natural numbers to define lists and other structures. Instead, however, we will use the Peano axioms as inspiration for axioms directly about lists. In other words, we will take lists as primitive objects and specify their properties by axioms similar to the Peano axioms. This includes induction axioms.

Our language for lists will include a constant $e$, denoting the empty list, and a binary function $\text{cons}$ that creates new lists out of previous ones. We intend that $\text{cons}(a,b)$ will mean the list with $a$ as its first element and $b$ as the remainder of the list.

Lists

We take the following set of axioms for basic lists.

List1: $\forall x \forall y \text{cons}(x, y) \neq e$.

List2: $\forall x \forall y \forall z \forall w (\text{cons}(x, y) = \text{cons}(z, w) \rightarrow (x = z \land y = w))$.

List3: For each formula $\varphi(x)$ and each variable $y$ not free in $\varphi$,

$$\varphi[e/x] \rightarrow (\forall x (\varphi \rightarrow \forall y \varphi[\text{cons}(y, x)/x]) \rightarrow \forall x \varphi)$$

Axioms List1 and List2 correspond to the first two Peano axioms: no $\text{cons}$ pair equals $e$, and equal $\text{cons}$ pairs have equal parts. Axiom List3 corresponds to Axiom PA7: it justifies the use of induction. Informally, it states that every object is constructed from $e$ by the use of $\text{cons}$.

Let's look at some of the objects that must appear in a domain, in order to satisfy these axioms. Aside from $e$, we must have $\text{cons}(e, e)$ (different than $e$, by List1). Thus we get a succession of objects:

$$\text{cons}(e, e), \text{cons}(e, \text{cons}(e, e)), \text{cons}(e, \text{cons}(e, \text{cons}(e, e))), \text{cons}(e, \text{cons}(e, \text{cons}(e, \text{cons}(e, e)))), \ldots$$

Each of these objects must be different, by List2.

We shall regard the above objects as lists whose elements are all $e$. In general, if we want a list containing $a_1, a_2, \ldots, a_k$ we use the object denoted by

$$\text{cons}(a_1, \text{cons}(a_2, \text{cons}(\ldots \text{cons}(a_k, e) \ldots)))$$

The values $a_i$ can be anything in the domain; in the case of basic lists, they must of course be lists themselves.

For example, the list containing the three objects $\text{cons}(e, e)$, $e$ and $\text{cons}(\text{cons}(e, e), e)$, in that order, is $\text{cons}(\text{cons}(e, e), \text{cons}(e, \text{cons}(\text{cons}(e, e), e)))$.

When lists get large, writing out the term in full can get cumbersome. To alleviate the problem somewhat, we adopt the following conventional notation using “angle brackets.”
• \(\langle\rangle\) denotes the empty list \(e\).

• For any object \(a\), \(\langle a\rangle\) denotes the list whose single item is \(a\), i.e., the object \(\text{cons}(a, e)\).

• For an object \(a\) and non-empty list \(\langle \ell \rangle\), \(\langle a, \ell \rangle\) denotes the list whose first item is \(a\) and whose remaining items are the items on the list \(\ell\). That is, \(\langle a, \ell \rangle\) denotes the list \(\text{cons}(a, \langle \ell \rangle)\).

3.1. Example.

1. Write the “angle bracket” form of the list \(\text{cons}(\text{cons}(e, e), \text{cons}(e, e), e)\).

2. Write the explicit term denoted by \(\langle e, e, e \rangle\).

3. Which list is longer: \(\langle e, e, e \rangle\) or \(\langle e, \langle e, e \rangle \rangle\)?

Answers:

1. The sub-term \(\text{cons}(e, e)\) is the one-element list \(\langle e \rangle\). The whole term contains that list twice; thus we denote it by \(\langle \langle e \rangle, \langle e \rangle \rangle\)

2. This is the list \(\text{cons}(e, \text{cons}(e, \text{cons}(e, e)))\) that we saw above.

3. The first list is longer. It has three items, while the second has only two.

3.2. Exercise. Prove that every non-\(e\) object is a \(\text{cons}\); that is, show that

\[\vdash \text{List} \forall x \; x \not= e \rightarrow \exists y \exists z \; \text{cons}(y, z) = x\,.

(Recall Exercise 2.2, that every non-zero natural number has a predecessor.)

To make use of these axioms, we shall add other symbols to the language: constants, functions and/or relations. When we do, Axiom List3 extends to formulas that include them.

Predicates and functions on lists

You already know a formalism for dealing with lists—Scheme programs have lists as a basic data type. Let's consider a Scheme program to append two lists, producing a third list.

```
(define (Append x y)
  (cond ((equal? x empty) y)
         (#t (cons (first x) (Append (rest x) y)))
    )
)
```

To use predicate logic to reason about a program such as \(\text{Append}\), we must understand the basic constructs of Scheme. The code above has five kinds of objects: variables, constants, relations, functions and control structures. Variables and constants translate easily: a Scheme variable or constant (\(x\), \(e\), etc.) simply becomes a variable or constant (\(x\), \(e\), etc.) Likewise the Scheme relation \(\text{equal}\?) corresponds to the relation =.

A control structure such as \(\text{cond}\) doesn't have a single equivalent; instead, it determines the course of the computation. The Scheme language defines \(\text{cond}\) such that \((\text{cond} \; (a \; b) \; c \ldots)\) evaluates to \(b\)
whenever a evaluates to #t. If a evaluates to #f, then the cond expression has the same evaluation as
(cond c...).

Functions pose a small problem. Scheme functions, such as first or Append, need not be defined
for all arguments. By contrast, function symbols in predicate logic create new terms, which must have
a value in every interpretation. Since we have no value for, say, first(e), we cannot simply use first as
a function symbol. There are several ways to deal with this issue. One that may look reasonable at
first is to introduce a new constant symbol “error”, and to use it for otherwise-undefined points. This,
however, becomes very awkward: for one thing, we must then account for “error” as a value in any
case. Further, programs have other ways to not produce a value than to encounter an error.

We shall therefore represent each desired partial function by an relation. Recall that a
k-ary partial
function p(x_1, x_2, ..., x_k) corresponds to the k + 1-ary relation R_p(x_1, x_2, ..., x_k, y) determined by the
rule that x_1, x_2, ..., x_k, y is in R_p if and only if y = p(x_1, x_2, ..., x_k).

We thus let R_first and R_rest be two binary relation symbols. We want “R_first(a, b)” to mean that
“the first of a is b”, and similarly for R_rest; therefore we take the axioms

\[ \forall x \forall y (R_{first}(x, y) \leftrightarrow \exists z \ x = \text{cons}(y, z)) \]

and

\[ \forall x \forall y (R_{rest}(x, y) \leftrightarrow \exists z \ x = \text{cons}(z, y)) \]

3.3. Exercise. For a relation symbol R, write sentences of predicate logic that characterize
1. whether R is the graph of a function, and
2. whether the function is total.

3.4. Exercise. Using the list axioms and the ones for R_first, prove the following.
1. Every item except empty has a first; that is, \( \forall x \ (x \neq \text{e} \rightarrow \exists y \ R_{first}(x, y)) \).
   (Hint: compare to Exercise 3.2.)
2. Any object x has at most one first: \( \forall x \forall y \ ((R_{first}(x, y) \land R_{first}(x, z)) \rightarrow y = z) \).

We can add a relation symbol for any Scheme function. For a built-in function, its definition in
Scheme determines the appropriate properties. For a defined function such as Append, our goal is to
find axioms that characterize the relation R_Append(x, y, z) so that it means “the result of (Append x y)
is z.”

To achieve this goal, let’s go through the definition of Append. The first line is

```
cond ( ( equal? x empty) y )
```

The definition of cond gives the formula \( x = e \rightarrow R_{Append}(x, y, y) \), which simplifies to

\[ R_{Append}(e, y, y) \].

For the remainder of the cond, we get the formula \( x \neq e \rightarrow \varphi \), where \( \varphi \) is the formula from the
second line:

\[ ^{8}\text{The relation } R_p \text{ is called the “graph” of the function } p. \text{ For the case } k = 1, \text{ you are quite familiar with drawing a graph of a function: the set of points you mark to draw a graph of } p \text{ is the relation } R_e. \]
cond (#t (cons (first x) (Append (rest x) y) )

Since the formula $\varphi$ must refer to the computed values of first, rest and Append, we use three new variables: $v_f$ for the value of first $x$, $v_r$ for the value of rest $x$, and $v_a$ for the value of (Append ...). The formula $\varphi$ is then

$$R_{\text{first}}(x, v_f) \rightarrow R_{\text{rest}}(x, v_r) \rightarrow R_{\text{Append}}(v_f, y, v_a) \rightarrow R_{\text{Append}}(x, y, \text{cons}(v_f, v_a))$$

Once again, our formula $x \neq e \rightarrow \varphi$ simplifies under the BL axioms. Since $x$ is not empty, it must be a cons (Exercise 3.2); then Axiom List2 yields $R_{\text{first}}(x, v_f) \rightarrow R_{\text{rest}}(x, v_r) \rightarrow x = \text{cons}(v_f, v_r)$. Substitution for $x$ (Axiom EQ2) then yields the formula

$$R_{\text{Append}}(v_r, y, v_a) \rightarrow R_{\text{Append}}(\text{cons}(v_f, v_r), y, \text{cons}(v_f, v_a))$$

In summary, the behaviour of the Scheme program Append is characterized by the following two formulas.

App1: $R_{\text{Append}}(e, y, y)$

App2: $R_{\text{Append}}(x, y, z) \rightarrow R_{\text{Append}}(\text{cons}(w, x), y, \text{cons}(w, z))$

3.5. Exercise. Prove that Append is total; that is, prove

$$\{\text{App1, App2}\} \vdash \text{List} \forall x \forall y \exists z R_{\text{Append}}(x, y, z)$$

3.6. Exercise.

1. Explain why the above formulas do not yield that $R_{\text{Append}}$ is functional; that is,

$$\{\text{App1, App2}\} \not\vdash \text{List} \forall x \forall y \exists z (R_{\text{Append}}(x, y, z) \land R_{\text{Append}}(x, y, w)) \rightarrow w = z$$

2. What property of the program Append (or indeed any Scheme program) did not get captured in the formulas above, but would allow a proof that $R_{\text{Append}}$ is functional?

3.7. Exercise. Show that the function Append is associative; that is, for all $x$, $y$ and $z$, the programs (Append $x$ (Append $y$ $z$) ) and (Append (Append $x$ $y$) $z$) produce the same result.

1. Give a formula, using the relation $R_{\text{Append}}$, that states the required property.

2. Show that your formula has a proof from the list axioms and App1 and App2.

4 Formulas for general Scheme programs

In this section, we shall describe how to construct a formula of predicate logic that describes the evaluation of any given Scheme program. We have two main tasks to accomplish.

Represent a program. In order to have a formula describe any property of programs, we must have an interpretation whose domain contains programs. We shall show how to represent a Scheme program as a list; thus we can use our familiar domain of lists.

Describe the execution of a program. The definition of the Scheme language provides “substitution” rules that transform one expression into another. To evaluate a program, an interpreter applies these rules successively until no rule applies to the latest expression. We shall show how to describe this process using predicate logic.

We shall discuss each in turn.
Representing expressions and programs

We start with a look back at our example program for Append.

```
(define (Append x y)
  ( cond ((equal? x empty) y)  
         (#t (cons (first x) (Append (rest x) y)) ) )
)
```

You'll recall how we treated the parts of this program before.

- We used variables for the values used in the program—the arguments \( x \) and \( y \) and some intermediate values that the program does not explicitly name.
- We used constants and relations for the built-in constants, functions, and relations (empty, rest, etc.) and also for the function Append itself.
- We did not directly represent control elements cond and define. Instead, we used cond to create the formulas linking the various parts, while define merely indicated that we had a function to represent.

To work with arbitrary programs, however, we must represent everything uniformly. In particular, we need the concept of a “name” of a value, function or the like, and a way to represent names so that formulas can refer to them.

What do we need from names? Not much. We need to be able to compare names, to determine whether or not they are the same. Also, we need to be able to have a “dictionary” of the meanings of names, so that we can look up a name and replace it with its meaning. To do these, we introduce a constant symbol name, and adopt the following convention.

A name is a list whose first is the constant name.

Thus we may express “\( x \) is a name” by the formula \( \exists y \ x = \text{cons}(\text{name}, y) \).

We assume a canonical way to transcribe text strings into names, and use the notation \( s \) to mean the name corresponding to the string \( s \). We require that if \( s \) and \( s' \) are different strings, then the corresponding names \( s \) and \( s' \) are also different, but we do not require other properties.\(^9\)

We do make a few exceptions to names, however. We shall represent the keywords cond, define, and lambda by their own constants,\(^10\) respectively cond, define, and \( \lambda \). Finally, empty and cons naturally become \( e \) and \( \text{cons} \), respectively.

With these conventions, we can now transcribe any Scheme expression into a term in the language of lists. For example, the program above becomes the term

\[
\langle \text{define}, \langle \text{Append}, x, y \rangle, \langle \text{cond}, \langle \langle \text{equal?}, x, e \rangle, y \rangle, \langle \#t, \langle \text{cons}, \langle \text{first}, x \rangle, \langle \text{Append}, \langle \text{rest}, x \rangle, y \rangle \rangle \rangle \rangle \rangle
\]

\(^9\)If we have constants \( a, b, \) etc. for text characters, then it seems natural to take \( abc \) to be \( \langle \text{name}, a, b, c \rangle \), and so on. However, we don't need this property; other conventions will do equally well.

\(^10\)We don't care what actual values these constants receive in an interpretation, as long as they don't clash with other things. Although the values must be lists (c.f. Exercise 3.2), any lists can be chosen.
Evaluation of full programs

(Note: this write-up may not match the overheads precisely. If they actually conflict, the overheads are more likely to be correct. If they are simply different, either version is likely equally adequate.)

Evaluation is the process of converting expressions to values. In Scheme, the basic step of evaluation is a substitution step: a replacement of one part of the expression by something else. If no substitution is possible, and the expression is a value, then the expression is fully evaluated. The rules specifying the allowed substitutions form part of the definition of the programming language. In our translation into predicate logic, we specify these rules by giving axioms.

We use a relation Step to describe the substitution process. It takes three arguments: a list representing the current state of execution, a list representing the dictionary of definitions of names, and a list representing a potential next state. Thus we want the term \( \text{Step}(x,D,y) \) to have value “true” if and only if expression \( x \) converts to expression \( y \) in one step, given dictionary \( D \). We shall enforce this condition by specifying axioms, where each part of the definition of Scheme becomes one or more axiom schemata.

For now, we shall make some simplifying assumptions, in order to present the main ideas without getting bogged down in details. Basically, we shall assume that the program never modifies a definition. Specifically, we shall assume

- all define statements come at the start of the program, with no two defining the same variable, and
- the program does not use set!, nor any other form of mutation.

Good programming practice, as you know, violates the first assumption by using local variables. However, given any program, one can modify it into one with no local variables without changing its behaviour in any way: simply re-name local variables so that they all have distinct names, and then make them global. Note that a program may still use recursion. If it does, then executing the program will assign a different value to the formal argument of the recursive function at each recursive invocation. We do allow this; we only forbid syntactic re-definition.

The prohibition of set! may seem very limiting, but ultimately it turns out not to pose a problem. We shall discuss this issue later, after doing the basic translation.

---

11 A substitution step is sometimes called a “rewrite”. These mean the same thing. Note that substitution in Scheme is essentially the same as the substitutions in formulas used in proofs.

12 In fact, an interpreter or compiler often makes such a modification “on the fly.” Two examples that you may find familiar:

- The definition of Scheme specifies that names be changed during execution. See, for example, the discussion of local definitions in Intermezzo 3 of How to Design Programs, by Felleisen, et al. Intermezzi 4 and 7 expand on the concept for expressions containing lambda and set!
- If you have used C++, Java, or other object-oriented languages, you have likely seen names like myType<int>(3) or <std::basic_ostream<char, std::char_traits<char>> > produced by the compiler or debugger. A programmer may write these (although need not), but they aren't the actual name used by the compiler. The actual name varies depending on the compiler and on the context in which the compiler chooses the name, but it might look something like

  _ZStl3IsSt11char_traitsIcEEERSt13basic_ostreamIcT_ES5_PKc.
Names

If a name denotes a built-in function, then we assume that the function is definable by a first-order relation. That is, there is a formula $\rho_b(\vec{x}, y)$ that is true if and only if $(b \vec{x})$ produces value $y$. For example, the formula $\rho_{\text{first}}$ for the built-in function $\text{first}$ is simply $R_{\text{first}}(x, y)$.

This leads to our first axiom schema for $\text{Step}$: for each built-in $b$,

$$\text{Step1: } \rho_b(\vec{x}, y) \rightarrow \text{Step}(\langle b, \vec{x} \rangle, D, y)$$

is an axiom.

If a name does not have a fixed definition specified by the language, we need to look it up in the dictionary. In terms of predicate logic, this means that we need a relation $\text{LookUp}$ such that $\text{LookUp}(x, D, y)$ evaluates to true if and only if dictionary $D$ specifies the value $y$ for the name $x$. But how do we specify such a relation? And what is a dictionary, anyway?

Abstractly, a dictionary is a mapping from names to values. Concretely, we shall use the standard data structure of an “association list”—implemented in predicate logic. You will recall that an association list is a list of pairs, where the first element of each pair is a name (or “index”) and the second element is its corresponding value (or definition). If we have a dictionary and a name to look up, there are only a few possibilities.

- If the first pair in the dictionary has the given name as its first element, then the desired value is the second element of the pair. This gives the axiom
  $$\text{Step2: } \text{LookUp}(x, \langle \langle x, y \rangle, z \rangle, y)$$

- If the first pair in the dictionary has something else as its first element, then the desired value is found by looking up the name in the rest of the dictionary. This gives the axiom
  $$\text{Step3: } x \neq u \rightarrow \text{LookUp}(x, z, y) \rightarrow \text{LookUp}(x, \text{cons}(\langle u, v \rangle, z), y)$$

- If the dictionary has no first pair—it is the empty list—no name has a value in the dictionary.

Once we have $\text{LookUp}$ characterized, we can use it for $\text{Step}$. We simply take the axiom

$$\text{Step4: } \text{LookUp}(x, D, y) \rightarrow \text{Step}(x, D, y)$$

Taking steps

There are several possible ways to take a step in evaluating an expression. One starts by looking at the first element of the expression. If the first element is an unevaluated expression, then take a step in that expression, leaving the rest unchanged. As an axiom, we get

$$\text{Step5: } \text{Step}(x, D, y) \rightarrow \text{Step}(\text{cons}(x, z), D, \text{cons}(y, z))$$

For example, if the expression is $\langle n, x \rangle$ for a name $n$, then the next step is $\langle v, x \rangle$, where $v$ is determined by either Axiom S1 (if $n$ is a built-in) or S4 (otherwise).

There are two other cases: that the first element is itself an evaluated expression (i.e., it is a value) and that the first element of the expression is not an expression at all, but rather a control element. We shall consider the latter case next.

We exemplify control elements using $\text{cond}$. Recall the definition of $(\text{cond} \ (a \ b) \ c \ldots)$: if $a$ evaluates to $\#f$, then $(\text{cond} \ (a \ b) \ c \ldots)$ has the same evaluation as $(\text{cond} \ c \ldots)$, while if $a$ evaluates to $\#t$, then $(\text{cond} \ (a \ b) \ c \ldots)$ has the same evaluation as $b$. The first condition corresponds to the axiom
Step 6: \( \text{Step}(\langle \text{cond}, \langle \#f, x \rangle, y \rangle, D, \langle \text{cond}, y \rangle) \)

and the second corresponds to

Step 7: \( \text{Step}(\langle \text{cond}, \langle \#t, x \rangle, y \rangle, D, x) \).

In the case that the guard is not a value, we evaluate it first:

Step 8: \( \text{Step}(z, D, w) \rightarrow \text{Step}(\langle \text{cond}, \langle z, x \rangle, y \rangle, D, \langle \text{cond}, \langle w, x \rangle, y \rangle) \).

We now turn to the case that the first element of an expression is a value.

Values

Scheme has many kinds of values: numbers, text strings, etc. Also, functions are values. Functions differ from other values, however, in that one can apply a function to arguments (other values), producing a result. This difference affects the evaluation of a sequence of values: if \( v \) and \( w \) are values, then \( (v \ w) \) may or may not be a value. If \( v \) is a number, then \( (v \ w) \) is a value—a list of two values. However, if \( v \) is a function of one argument, then \( (v \ w) \) is not a value: one needs to apply the function in order to evaluate the expression. Values that are not functions we shall call “inert”; let \( \text{IsInert}(x) \) denote the formula \( \text{IsValue}(x) \land \neg \text{R}_{\text{first}}(x, \lambda) \).

- Each Scheme constant \( c \) is a value. (For example, \( c \) might be empty.)

  Step 9: \( \text{IsValue}(c) \).

- A lambda-expression is a value:

  Step 10: \( \forall x \forall y \text{IsValue}(\langle \lambda, \langle \text{name}, x \rangle, y \rangle) \).

- If \( x \) and \( y \) are values, and \( x \) is inert, then \( \text{cons}(x, y) \) is a value:

  Step 11: \( \text{IsValue}(y) \rightarrow \text{IsInert}(x) \rightarrow \text{IsValue}(\text{cons}(x, y)) \).

Note that names are not values. A term of the form \( \text{cons}(\text{name}, x) \) can be evaluated by looking \( x \) up in the dictionary.

Applying functions

We now reach the final case of taking a step in evaluating an expression: the application of a function. The basic definition in Scheme works by substitution: to apply a function \( \lambda \text{mbda}(x) \ y \) to a value \( u \), substitute \( u \) for the name \( x \) everywhere \( x \) occurs in the expression \( y \). This gives the axiom

Step 12: \( \text{IsValue}(u) \rightarrow \text{Subst}((x, u), t, v) \rightarrow \text{Step}(\text{cons}((\lambda, x, t), u), D, v)) \)

The implicant \( \text{IsValue}(u) \) appears because Scheme specifies that functions may only be applied to evaluated arguments.\(^{13}\) To handle unevaluated arguments, we simply evaluate them:

\(^{13}\)This marks one significant difference between Scheme and other forms of Lisp, which allow unevaluated terms as arguments. In fact, “pure” Lisp requires that one apply a function before evaluating its arguments. This order of evaluation has the advantage that some programs terminate that would not terminate with the argument-first order. The results, however, can prove very surprising to programmers—especially if they use local variables.
Step 13: \( \text{Step}(u, D, v) \rightarrow \text{Step}(\text{cons}(\langle \lambda, x, t \rangle, u), D, \text{cons}(\langle \lambda, x, t \rangle, v)) \).  

This leaves us with the substitution itself, which is a relatively straightforward case of inductive definition. We have the base case

Step 14: \( \text{Subst}(\langle x, u \rangle, e, e) \)

and the inductive case

Step 15: \( \text{Subst}(\langle x, u \rangle, t, v) \rightarrow \text{Subst}(\langle x, u \rangle, y, z) \rightarrow \text{Subst}(\langle x, u \rangle, \langle y, t \rangle, \langle z, v \rangle). \)
5 What Scheme can’t do

We now turn to considering some limitations of Scheme. We shall then return to predicate logic, and show that it has the same limitations.

Testing Whether a Program Halts

Some Scheme programs terminate after a finite number of steps; others do not. For example, consider the following.

```scheme
(define (loop x) (loop loop))
```

With this definition, the substitution rule never makes any progress:

```
(loop loop) ⇒ (loop loop)
⇒ (loop loop)
⇒ ...
```

Can we distinguish between programs that halt and those that don’t? Sometimes we can, of course. But can we always do it? More precisely, can we write a Scheme function `halts?` that determines whether its argument—another function—will halt on a given input? That is, we would like to have a function `halts?` that meets the following specification.

```
;; Contract: halts? : SchemeProgram Input → boolean
;; If the evaluation of ( P I ) halts, then (halts? P I) halts with value #t, and
;; If the evaluation of ( P I ) does not halt, then (halts? P I) halts with value #f.
;; Example: ( halts? loop loop ) returns #f.
```

It turns out that no such program exists. A program can do a lot towards “understanding” another program, but not everything.

5.1. Theorem. No Scheme function can perform the task required of `halts?`, correctly for all programs.

To prove this result, we argue by contradiction. Suppose that someone claims to have a `halts?` function that meets the condition required above. By careful argument, we can show that their function fails to do the job. We won’t actually analyze it directly; instead, we will write new functions that make use of it.

First, we copy their function:

```
(define (halts? P I) ...)
```

Next, we consider creating other functions that make use of `halts?`. For example, we can define a function that calls `halts?` with both arguments being the same function.

```
(define (self-halt? P) (halts? P P))
```

14Don’t try this at school—it constitutes a denial-of-service attack!
This should answer the question, “does P terminate when given itself as input?”

What does self-halt? do when given itself as input? In other words, what’s the result of the
invocation (self-halt? self-halt?)? Let’s see.

\[
\begin{align*}
&\Rightarrow \ldots \quad \text{; evaluation of halts? -- which must halt} \\
&\Rightarrow \begin{cases} 
#t, \text{ if (self-halt? self-halt?) halts,} \\
#f, \text{ if (self-halt? self-halt?) doesn’t halt.}
\end{cases}
\end{align*}
\]

Since halts? always terminates, the evaluation of (self-halt? self-halt?) also terminates. And, since halts? gives the correct answer, the final result must be #t.

So far, so good—if a bit strange. But we can take it another step. Consider the function

\[
\begin{align*}
( \text{define (halt-if-loops P)}) &\Rightarrow (\text{cond [ (halts? P P) (loop loop)]} \\
&\quad \text{[ else #t]}) \\
&\Rightarrow \ldots \quad \text{; evaluation of halts? -- which must halt} \\
&\Rightarrow \begin{cases} 
(\text{loop loop}), \text{ if (halt-if-loops halt-if-loops) halts,} \\
#t, \text{ if (halt-if-loops halt-if-loops) doesn’t halt.}
\end{cases}
\end{align*}
\]

The evaluation of the program (halt-if-loops halt-if-loops) terminates if and only if evaluation of the program (halt-if-loops halt-if-loops) doesn’t terminate. Impossible! No such program exists.

We made only one assumption: that the original halts? function worked correctly. Thus that assumption must be false: the halts? function we started with does not work correctly.

Thus we have proven the theorem: no Scheme function can correctly test whether a given program terminates on a given input.
Other Undecidable Problems

We define two computational problems.

PROVABILITY
Given a formula \( \varphi \) of predicate logic, does \( \varphi \) have a proof?

INTEGERROOT
Given a polynomial \( q(x_1, x_2, \ldots, x_n) \) with integer coefficients, does \( q \) have an integral root; that is, are there integers \( a_1, a_2, \ldots, a_n \) such that \( q(a_1, a_2, \ldots, a_n) = 0 \)?

5.2. Theorem.

A. No algorithm can solve problem PROVABILITY, correctly in all cases.

B. No algorithm can solve problem INTEGERROOT, correctly in all cases.

Both proofs follow the same basic plan. We start with PROVABILITY. The proof has two steps.

1. Devise an algorithm to solve the following problem.
   
   Given a program \( (P I) \), produce a formula \( \varphi_{P,I} \) such that
   
   \( \varphi_{P,I} \) has a proof \( \iff \) \( (P I) \) halts.

2. If some algorithm solves PROVABILITY, then we can combine it with the above algorithm to get an algorithm that solves HALTING. But no algorithm solves HALTING.

Therefore, no algorithm solves PROVABILITY.

Similarly, for INTEGERROOT:

1. Devise an algorithm to solve the following problem.
   
   Given a program \( (P I) \), produce a polynomial \( q_{P,I} \) such that
   
   \( q_{P,I} \) has an integral root \( \iff \) \( (P I) \) halts.

2. If some algorithm solves INTEGERROOT, then we can combine it with the above algorithm to get an algorithm that solves HALTING. But no algorithm solves HALTING.

Therefore, no algorithm solves INTEGERROOT.
6 Undecidability and Incompleteness

We have seen that no Scheme program can, in all cases, test whether a program given as input halts on a specified input. How much “power” would some other formalism require, in order to express this halting property? Perhaps first-order logic might suffice?

6.1. Lemma. There is a Scheme program that, given a well-formed formula $\varphi$, outputs a proof of $\varphi$ if one exists. If no proof exists, the program may run forever, with no output.

Proof sketch. Consider a program that generates a sequence of formulas, and then checks whether the sequence is actually a correct proof of $\varphi$. If so, it outputs the sequence. Otherwise, it starts over with another sequence.

If the program generates the sequences in a suitable order, then every possible sequence will appear eventually. Thus if any proof of $\varphi$ exists, the program will eventually examine it, and then output it. □

6.2. Theorem (Gödel’s Incompleteness Theorem). Let $\Gamma$ be a set of formulas, such that membership in $\Gamma$ is decidable; that is, there is a Scheme program that with a formula $\varphi$ as input, outputs “true” if $\varphi \in \Gamma$ and outputs “false” if $\varphi \notin \Gamma$. Then there are two cases: either

1. $\Sigma_{\text{List}} \cup \Gamma$ is inconsistent, or
2. There is a formula $\varphi$ such that $\Sigma_{\text{List}} \cup \Gamma \not\vdash \varphi$ and $\Sigma_{\text{List}} \cup \Gamma \not\vdash \neg \varphi$.

(In the theorem, $\Sigma_{\text{List}}$ denotes the set \{List1,List2,List3\} of axioms for lists.)

Proof. Suppose that $\Sigma_{\text{List}} \cup \Gamma$ is consistent. Consider a Scheme program that operates as follows.

On input $S$, $x$:

Let $\eta$ denote the formula $\exists y (\text{Eval}(S, x, y) \land \text{final}(y))$.
Search for a proof of either $\Sigma_{\text{List}} \cup \Gamma \vdash \eta$ or $\Sigma_{\text{List}} \cup \Gamma \vdash \neg \eta$.
If a proof of $\eta$ is found, halt with output “$S$ halts on input $x$”.
If a proof of $\neg \eta$ is found, halt output “$S$ does not halt on input $x$”.

We argue that this program cannot halt on all formulas. If it did, it would decide the halting problem for Scheme programs—but no such program exists.

Thus there must be some $S$ and $x$ such that the program does not halt on inputs $S$ and $x$. Therefore, the formula $\eta(S, x)$ is an example of a formula that meets the required condition—neither proof $\Sigma_{\text{List}} \cup \Gamma \vdash \eta(S, x)$ nor $\Sigma_{\text{List}} \cup \Gamma \vdash \neg \eta(S, x)$ exists. □