Logic and Computation:

From Arithmetic to Programs

J. Buss

with formatting assistance from D. Maftuleac

Contents

- Arithmetic
- Lists
- Programs

Axioms

Thus far, we have used symbols for functions, relations, etc., without a fixed meaning. An interpretation can specify them arbitrarily.

Often, however, we want to use one or more symbols in a specific way. The usual approach to this is to use axioms.

> *Definition:* An *axiom* is a formula that is assumed as a premise in any proof. An *axiom schema* is a set of axioms, defined by a pattern or rule.

Axioms often behave like additional "inference rules".

Example: Axioms for equality

Earlier, we defined Predicate Logic with Equality by inference rules:

$$\frac{}{t = t}\ =\text{i} \qquad \frac{t_1 = t_2 \quad \alpha[t_1/z]}{\alpha[t_2/z]}\ =\text{e} \ .$$

Instead of those rules, we could take the following axiom and schema.

EQ1: $\forall x \cdot x = x$ is an axiom.

EQ2: $\forall x_1 \cdot \forall x_2 \cdot \big(x_1 = x_2 \to \big(\alpha[x_1/z] \to \alpha[x_2/z]\big)\big)$ is an axiom, for any choice of formula α and variable z.

Using $\forall$e and $\to$e on these axioms permits the same deductions that rules =i and =e do.

Why Use Axioms?

Why use axioms?

- Axioms constrain possible interpretations.
- Some axioms don't have a simple corresponding rule.
- In a computer implemention, inference rules get built in. Thus new inference rules require rewriting code. Axioms, on the other hand, are easy to add — or to remove.

Natural Numbers

Fix the domain as $\mathbb{N}$, the natural numbers. Interpret the constant symbol 0 as zero and the unary function symbol s as successor.

Thus each number in $\mathbb{N}$ has a term: 0, $s(0)$, $s(s(0))$, $s(s(s(0)))$, ...

Zero and successor satisfy the following axioms.

PA1: $\forall x \cdot s(x) \neq 0$.
"Zero is not a successor."

PA2: $\forall x \cdot \forall y \cdot \big(s(x) = s(y) \to x = y\big)$.
"Nothing has two predecessors."

("PA" stands for Peano Axioms, named for Giuseppe Peano.)

Addition and Multiplication

Further axioms characterize + (addition) and × (multiplication).

PA3: $\forall x \cdot (x + 0 = x)$.
Adding zero to any number yields the same number.

PA4: $\forall x \cdot \forall y \cdot x + s(y) = s(x + y)$.
Adding a successor yields the successor of adding the number.

PA5: $\forall x \cdot x \times 0 = 0$.
Multiplying by zero yields zero.

PA6: $\forall x \cdot \forall y \cdot (x \times s(y) = x \times y + x)$.
Multiplication by a successor.

Induction in Peano Arithmetic

The six axioms above define $+$ and $\times$ for any particular numbers.
They do not, however, allow us to reason adequately about all numbers.
For that, we use an additional axiom: induction.

PA7: For each formula φ and variable x,

$$\varphi[0/x] \to \Big(\forall x \cdot \big(\varphi \to \varphi[s(x)/x]\big) \to \forall x \cdot \varphi\Big)$$

is an axiom.

The formula φ represents the "property" to be proved.

To prove φ for every x, we can prove the base case $\varphi[0/x]$
and the inductive case $\forall x \cdot (\varphi \to \varphi[s(x)/x])$.

Properties from the Peano Axioms

These axioms imply all of the familiar properties of the natural numbers.

For example, addition is commutative.

Theorem: Addition in Peano Arithmetic is commutative; that is,

$$\vdash_{PA} \forall x \cdot \forall y \cdot (x + y = y + x) \ .$$

(Notation "$\vdash_{PA}$" means "provable in Natural Deduction using Axioms EQ1–2 and PA1–7".)

How can we find such a proof?

We must use induction (PA7). The key first step: choose a good formula φ for the induction property.

Setting Up the Induction

Recall Axiom PA7: $\varphi[0/x] \to \Big(\forall x \cdot \big(\varphi \to \varphi[s(x)/x]\big) \to \forall x \cdot \varphi\Big)$

Choosing φ to be $\forall y \cdot (x + y = y + x)$ yields the instance

$$\begin{aligned}
&\forall y \cdot (0 + y = y + 0) \to \\
&\quad \Big(\forall x \cdot \big(\forall y \cdot (x + y = y + x) \to \forall y \cdot (s(x) + y = y + s(x))\big) \to \\
&\qquad\qquad \forall x \cdot \forall y \cdot (x + y = y + x)\Big)
\end{aligned}$$

Thus we must prove the base case $\forall y \cdot (0 + y = y + 0)$
and the inductive case

$$\forall x \cdot \big(\forall y \cdot (x + y = y + x) \to \forall y \cdot (s(x) + y = y + s(x))\big) \ .$$

The Base Case for Commutativity

To prove: $\forall y \cdot 0 + y = y + 0$.

How?

The Base Case for Commutativity

To prove: $\forall y \cdot 0 + y = y + 0$.

How?

We must use induction, in order to prove the base case.
To control the complication, let's make it a lemma:

> *Lemma*
> Peano Arithmetic has a proof of $\forall y \cdot 0 + y = y + 0$.

Plan, to prove lemma: induction (PA7) with $0 + y = y + 0$ for φ.

Target $0 + 0 = 0 + 0$: from EQ1.

Lemma, continued

Target $\forall y \cdot \big((0 + y = y + 0) \to (0 + s(y) = s(y) + 0)\big)$:

On the assumption $0 + y' = y' + 0$, we get

$$\begin{aligned} 0 + s(y') &= s(0 + y') && \text{PA4} \\ &= s(y' + 0) && \text{assumption} + \text{EQsubs}(s(\cdot)) \\ &= s(y') + 0 && \text{PA3.} \end{aligned}$$

Then $\to$i and generalization ($\forall$i) yield the target.

Using PA7 and $\to$e (twice) completes the proof of the lemma.

The full proof of the base case appears next.

Commutativity of Addition: The base case

1.	$0+0=0+0$	EQ1 + ∀e
2.	y fresh	
3.	$0+y=y+0$	Assumption
4.	$0+s(y)=s(0+y)$	PA4 + ∀e
5.	$s(0+y)=s(y+0)$	EQsubs($s(\cdot)$): 3
6.	$y+0=y$	PA3 + ∀e
7.	$s(y+0)=s(y)$	EQsubs($s(\cdot)$): 6
8.	$s(y)+0=s(y)$	PA3 + ∀e
9.	$0+s(y)=s(y)+0$	EQtrans(3): 4, 5, 7, 8
10.	$0+y=y+0 \rightarrow 0+s(y)=s(y)+0$	→i: 3–9
11.	$\forall y \cdot \big(0+y=y+0 \rightarrow 0+s(y)=s(y)+0\big)$	∀i: 10
12.	$\forall y \cdot 0+y=y+0$	PA7 + ∀e + →e(×2): 1, 11

Inductive Case for Commutativity

Lemma. For each free variable x,

$$\forall y \cdot (x + y = y + x) \vdash_{PA} \forall y \cdot (s(x) + y = y + s(x)) \ .$$

Plan of proof: induction on variable y for formula $s(x) + y = y + s(x)$.

Target $s(x) + 0 = 0 + s(x)$: as before.

Target $\forall z \cdot \big((s(x) + z = z + s(x)) \to (s(x) + s(z) = s(z) + s(x))\big)$:

Assuming $s(x) + z' = z' + s(x)$ yields

$$\begin{aligned} s(x) + s(z') &= s(s(x) + z') && \text{PA4} \\ &= s(z' + s(x)) && \text{assumption} + \text{EQsubs}(s(\cdot)) \\ &= s(s(z' + x)) && \text{PA4} + \text{EQsubs}(s(\cdot)). \end{aligned}$$

Inductive Case, continued

The premise of the lemma implies $x + s(z') = s(z') + x$; thus

$$\begin{aligned} s(z') + s(x) &= s(s(z') + x) && \text{PA4} \\ &= s(x + s(z')) && \text{premise} + \text{EQsubs}(s(\cdot)) \\ &= s(s(x + z')) && \text{PA4} + \text{EQsubs}(s(\cdot)). \end{aligned}$$

Since $x + z' = z' + x$ by the premise, EQsubs$(s(s(\cdot)))$ yields

$$s(x) + s(z') = s(z') + s(x) \ ,$$

as required.

Using PA7 and $\rightarrow$e (twice) completes the proof of the lemma.

(Note that we had to use $\forall$e on the premise twice, with different terms.)

1.	$\forall y \cdot x + y = y + x$	Premise
2.	$s(x) + 0 = 0 + s(x)$	Lemma, p. 248, + ∀e
3.	$x + z = z + x$, z fresh	∀e: 1
4.	$s(x) + z = z + s(x)$	Assumption
5.	$x + s(z) = s(z) + x$	∀e: 1
6.	*(Uses of PA4 omitted)*	
10.	*(Uses of EQsubs on 3–7 omitted)*	
15.	$s(x) + s(z) = s(z) + s(x)$	EQtrans(6): 8–14
16.	$s(x) + z = z + s(x) \rightarrow s(x) + s(z) = s(z) + s(x)$	→i: 4–15
17.	$\forall z \cdot \big((s(x) + z = z + s(x)) \rightarrow (s(x) + s(z) = s(z) + s(x))\big)$	∀i: 16
18.	$\forall y \cdot s(x) + y = y + s(x)$	PA7 + ∀e + →e(×2): 2, 17

Putting It All Together

1.	$0+0=0$	PA3 + ∀e
2.	*Lemma, p. 248*	
14.	$\forall y \cdot 0+y=y+0$	PA7 + →e(×2): 1, 12
15.	$\forall y \cdot x+y=y+x$	Assumption
16.	*Lemma, p. 251*	
33.	$\forall y \cdot x+y=y+x \rightarrow \forall y \cdot s(x)+y=y+s(x)$	→i: 15–32
34.	$\forall x \cdot \forall y \cdot x+y=y+x$	PA7 + ∀e + →e(×2): 14, 33

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

Definability

Let formula φ have free variables $x_1,\ldots,x_k$.

Given an interpretation $\mathcal{I}$, a formula φ *defines* the k-ary relation of tuples that make φ true — that is, the relation

$$R_\varphi = \left\{ \langle \mathrm{a}_1,\ldots,\mathrm{a}_\mathrm{k}\rangle \in dom(I)^k \mid \varphi^{(\mathcal{I},\theta[x_1\mapsto \mathrm{a}_1]\cdots[x_k\mapsto \mathrm{a}_k])} = \mathrm{T} \right\} \quad .$$

A relation R is *definable (in $\mathcal{I}$)* iff $R = R_\varphi$ for some formula φ.

Example: in Peano Arithmetic, the relation $\leq$ is defined by the formula

$$\exists z \cdot x_1 + z = x_2 \quad .$$

Properties of Defined Relations

The PA axioms allow one to show that the defined relation $\leq$ has the usual properties.

- $x \leq y$ and $y \leq z$ imply $x \leq z$ (transitivity).
- If $x \leq y$ and $y \leq x$ then $x = y$.

We can also define further relations using $\leq$; e.g.,

$$x < y \text{ iff } (x \leq y) \wedge x \neq y \quad .$$

Transitivity of Less-or-Equal

To show: $x \leq y,\ y \leq z \vdash x \leq z$.

1.	$\exists w \cdot x + w = y$	Premise
2.	$\exists w \cdot y + w = z$	Premise
3.	$x + u = y$, u fresh	Assumption
4.	$y + v = z$, v fresh	Assumption
5.	$x + (u + v) = (x + u) + v$	Associativity of $+$
6.	$(x + u) + v = y + v$	EQsubs$(w + v)$: 3
7.	$x + (u + v) = z$	EQtrans(2): 5, 6, 4
8.	$\exists w \cdot x + w = z$	$\exists$i: 7
9.	$\exists w \cdot x + w = z$	$\exists$e: 2, 4–8
10.	$\exists w \cdot x + w = z$	$\exists$e: 1, 3–9

Defining Functions

To define a k-ary function, use its $(k+1)$-ary relation.

Example: Let R_{sq} ("square-of") be defined by $x_1 \times x_1 = x_2$.
Then we can get the effect of having the squaring function:

> if φ contains a free variable x, but u is fresh, then the formula
>
> $$\exists u \cdot (R_{sq}(t,u) \wedge \varphi[u/x])$$
>
> expresses "the square of t satisfies φ."

We must, however, ensure that R_{sq} really does define a function; that is, every number has exactly one square:

$$\forall x \cdot \Big((\exists y \cdot R_{sq}(x,y)) \wedge \forall y \cdot \forall z \cdot \big((R_{sq}(x,y) \wedge R_{sq}(x,z)) \rightarrow y = z \big) \Big) \ .$$

We leave this proof as an exercise.

Lists

Lists

Lists are a basic structure in computer science. We shall

- Give axioms for lists, similar to the PA axioms.
- Use lists to describe Scheme programs.

Two views:

- "Basic lists", which have only lists, and
- "General lists", which allow other objects as items on the list.

Vocabulary for lists:

- a constant symbol e, for the empty list, and
- a binary function symbol *cons*.

A term $cons(a, b)$ will denote the list with a as its first element and b as the remainder of the list.

Lists and Natural Numbers

We define lists by an analogy to the natural numbers.

Recall that the natural numbers start with the constant $0 \in \mathbb{N}$ and use the successor function s to generate any natural number.

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

Similarly, **basic lists** start with the empty list e, and the *cons* function generates new lists out of previous ones.

$$e, \mathit{cons}(e, e), \mathit{cons}(e, \mathit{cons}(e, e)), \mathit{cons}(e, \mathit{cons}(e, \mathit{cons}(e, e))), \ldots \quad .$$

We also get other objects, when the first argument to *cons* is not e.

What's a List?

A domain of lists must contain the empty list e.
It must also contain $cons(e, e)$, and so on:

$cons(e, cons(e, e))$,
$cons(e, cons(e, cons(e, e)))$, ...

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 formed as

$$cons(a_1, cons(a_2, cons(\ldots cons(a_k, e)\ldots))) \ .$$

The values a_i can be anything in the domain.

In the case of basic lists, they are lists themselves.

Examples

For example, the list containing the three objects $cons(e, e)$, e and $cons(cons(e, e), e)$, in that order, is

$$cons\big(cons(e, e), cons\big(e, cons(cons(cons(e, e), e), e)\big)\big) \quad .$$

A short-hand notation: "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 $cons(a, e)$ (which is also $cons(a, \langle\rangle)$).
- 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 $\langle\ell\rangle$. That is, $\langle a, \ell\rangle$ denotes the list $cons(a, \langle\ell\rangle)$.

Note: $\langle a, \ell\rangle$ is NOT the same as $\langle a, \langle\ell\rangle\rangle$!

Examplecises

1. What is the "angle bracket" form of the list $cons(\, cons(e, e), cons(\, cons(e, e), e))$?

Examplecises

1. What is the "angle bracket" form of the list
   $cons(\, cons(e, e), cons(\, cons(e, e), e))$?

   The sub-term $cons(e, e)$ is the one-element list $\langle e \rangle$.
   The whole term contains that list twice, yielding $\langle \langle e \rangle , \langle e \rangle \rangle$.

Examplecises

1. What is the "angle bracket" form of the list
   $cons(\, cons(e, e), cons(\, cons(e, e), e))$?

   The sub-term $cons(e, e)$ is the one-element list $\langle e \rangle$.
   The whole term contains that list twice, yielding $\langle \langle e \rangle , \langle e \rangle \rangle$.

2. What is the explicit term denoted by $\langle e, e, e \rangle$?

Examplecises

1. What is the "angle bracket" form of the list $cons(\, cons(e, e), cons(\, cons(e, e), e))$?

   The sub-term $cons(e, e)$ is the one-element list $\langle e \rangle$.
   The whole term contains that list twice, yielding $\langle \langle e \rangle , \langle e \rangle \rangle$.

2. What is the explicit term denoted by $\langle e, e, e \rangle$?

   The list $cons(e, cons(e, cons(e, e)))$ that we saw above.

Examplecises

1. What is the "angle bracket" form of the list $cons(\, cons(e, e), cons(\, cons(e, e), e))$?

   The sub-term $cons(e, e)$ is the one-element list $\langle e \rangle$.
   The whole term contains that list twice, yielding $\langle \langle e \rangle , \langle e \rangle \rangle$.

2. What is the explicit term denoted by $\langle e, e, e \rangle$?

   The list $cons(e, cons(e, cons(e, e)))$ that we saw above.

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

Examplecises

1. What is the "angle bracket" form of the list $cons(\, cons(e, e), cons(\, cons(e, e), e))$?

   The sub-term $cons(e, e)$ is the one-element list $\langle e \rangle$.
   The whole term contains that list twice, yielding $\langle \langle e \rangle , \langle e \rangle \rangle$.

2. What is the explicit term denoted by $\langle e, e, e \rangle$?

   The list $cons(e, cons(e, cons(e, e)))$ that we saw above.

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

   The first list is longer. It has three items; the second only two.

Axioms of Basic Lists

We take the following set of axioms for basic lists.

BL1: $\forall x \cdot \forall y \cdot cons(x, y) \neq e$.

BL2: $\forall x \cdot \forall y \cdot \forall z \cdot \forall w \cdot \big(cons(x, y) = cons(z, w) \rightarrow (x = z \wedge y = w)\big)$.

BL3: For each formula $\varphi(x)$ and each variable y not free in φ,

$$\varphi[e/x] \rightarrow \big(\forall x \cdot \big(\varphi \rightarrow \forall y \cdot \varphi[cons(y, x)/x]\big) \rightarrow \forall x \cdot \varphi\big)$$

Note the close analogy to natural numbers!

- e corresponds to 0.
- *cons* is a binary analogue of s.
- An induction axiom applies.

Exercise: Reasoning about lists

Exercise:

Prove that every non-e object is a *cons*; that is, show that

$$\vdash_{BL} \forall x \cdot \big(x \neq e \rightarrow \exists y \cdot \exists z \cdot \mathit{cons}(y,z) = x\big) \ .$$

(Hint: the corresponding statement for natural numbers is that every non-zero number is a successor:

$$\vdash_{PA} \forall x \cdot \big(x \neq 0 \rightarrow \exists z \cdot s(z) = x\big) \ .)$$

Relations on Lists

We can define relations and functions on lists.

Example: the function $\mathit{first}(\cdot)$, where the first item on list x is $\mathit{first}(x)$.

BUT: That's not a function! The list e has no first.

Relations on Lists

We can define relations and functions on lists.

Example: the function *first*$(\cdot)$, where the first item on list x is *first*(x).

BUT: That's not a function! The list e has no first.

Solution: use the relation

$$\mathcal{R}_{\texttt{first}} = \left\{ \left(\mathit{cons}(\mathrm{a}, \mathrm{b}), \mathrm{a} \right) \mid \mathrm{a} \text{ and } \mathrm{b} \text{ are lists} \right\} \quad .$$

$\mathcal{R}_{\texttt{first}}(x, y)$ is definable by the formula $\exists z \cdot x = \mathit{cons}(y, z)$.

Relations on Lists

We can define relations and functions on lists.

Example: the function *first*$(\cdot)$, where the first item on list x is *first*(x).

BUT: That's not a function! The list e has no first.

Solution: use the relation

$$\mathcal{R}_{\texttt{first}} = \left\{ \left(\mathit{cons}(\mathrm{a}, \mathrm{b}), \mathrm{a} \right) \mid \mathrm{a} \text{ and } \mathrm{b} \text{ are lists} \right\} \quad .$$

$\mathcal{R}_{\texttt{first}}(x, y)$ is definable by the formula $\exists z \cdot x = \mathit{cons}(y, z)$.

Similarly, formula $\exists z \cdot x = \mathit{cons}(z, y)$ defines

$$\mathcal{R}_{\texttt{rest}}(x, y) = \left\{ \left(\mathit{cons}(\mathrm{a}, \mathrm{b}), \mathrm{b} \right) \mid \mathrm{a} \text{ and } \mathrm{b} \text{ are lists} \right\} \quad .$$

Properties of "First" and "Rest"

Lemma

Any object has at most one first:

$$\forall x \cdot \forall y \cdot \forall z \cdot \Big((\mathcal{R}_{\texttt{first}}(x,y) \wedge \mathcal{R}_{\texttt{first}}(x,z)) \rightarrow y = z \Big) \ .$$

Sketch of proof: Assume $\mathcal{R}_{\texttt{first}}(x,y) \wedge \mathcal{R}_{\texttt{first}}(x,z)$; that is,

$$\exists u \cdot x = \mathit{cons}(y,u) \quad \text{and} \quad \exists u \cdot x = \mathit{cons}(z,u) \ .$$

Thus there are items u_1 and u_2 (rule $\exists$e) for which

$$x = \mathit{cons}(y,u_1) \quad \text{and} \quad x = \mathit{cons}(z,u_2) \ .$$

Then BL2 yields $y = z$.

Length of Lists

Example: Constrain a relation *EqLen* so that *EqLen*(x, y) means that x and y have the same length.

Length of Lists

Example: Constrain a relation *EqLen* so that $EqLen(x, y)$ means that x and y have the same length.

- The empty list has the same length as itself: $EqLen(e, e)$.

Length of Lists

Example: Constrain a relation *EqLen* so that $EqLen(x, y)$ means that x and y have the same length.

- The empty list has the same length as itself: $EqLen(e, e)$.
- A non-empty list does not have the same length as the empty list:

$$\forall x \cdot \big(x \neq e \rightarrow (\neg\, EqLen(x, e) \wedge \neg\, EqLen(e, x)\big) \ .$$

Length of Lists

Example: Constrain a relation *EqLen* so that $EqLen(x, y)$ means that x and y have the same length.

- The empty list has the same length as itself: $EqLen(e, e)$.
- A non-empty list does not have the same length as the empty list:

$$\forall x \cdot \big(x \neq e \rightarrow (\neg\, EqLen(x, e) \wedge \neg\, EqLen(e, x)\big) \ .$$

- Adding an element to equal-length lists produces equal-length lists:

$$\forall x \cdot \forall y \cdot \forall u \cdot \forall v \cdot \big(EqLen(x, y) \longleftrightarrow EqLen(cons(u, x), cons(v, y))\big) \ .$$

Length of Lists, continued

To show:

Every list has the same length as itself; that is, $\forall x \cdot EqLen(x, x)$.

Length of Lists, continued

To show:

Every list has the same length as itself; that is, $\forall x \cdot EqLen(x, x)$.

We prove this using induction, via BL3.

Basis: $\vdash_{BL} EqLen(e, e)$. Given.

Inductive step: Need to prove, for arbitrary ("fresh") x, that

$$EqLen(x, x) \vdash_{BL} \forall y \cdot EqLen(cons(y, x), cons(y, x)) \ .$$

This follows directly from the third formula for *EqLen*.

Exercise on Defined Relations

Similarly to *EqLen*, we can characterize *UnEqLen*, "unequal lengths", by

$$\neg\, \mathit{UnEqLen}(e, e) \ ,$$

$$\forall x \cdot (x \neq e \rightarrow \mathit{UnEqLen}(x, e)) \ ,$$

and

$$\forall x \cdot \forall y \cdot \forall u \cdot \forall v \cdot \big(\mathit{UnEqLen}(x, y) \longleftrightarrow \mathit{UnEqLen}(\mathit{cons}(u, x), \mathit{cons}(v, y))\big) \ .$$

Exercise: Show that $\vdash_{BL} \forall x \cdot \forall y \cdot \big(\mathit{UnEqLen}(x, y) \longleftrightarrow \neg\, \mathit{EqLen}(x, y)\big)$.

Programs Using Lists

Computer programs often use lists.

We can analyze such programs via predicate logic.

A Scheme Function Using Lists

Suppose we have a Scheme function: `( define (Append x y) ... )`

Our goal: to describe, in FOL, a function $f(x, y)$ that yields the result of `(Append x y)`. We can then reason about f.

Approach: Let $\mathcal{S}$ be an interpretation whose domain is the set of Scheme values (i.e., lists). It interprets the constants, relations, etc., as specified for Scheme.

A call to function `Append`, with value a for x and b for y, creates an environment θ with $\theta(x) = a$ and $\theta(y) = b$. Thus we want the value $f(x, y)^{(\mathcal{S},\theta)}$ to be the return value of `Append`.

However, there is a small-but-significant problem. . . .

Partial Vs. Total Functions

When discussing Scheme programs, we use the word "function" to include *partial* functions—ones that lack a value for some arguments.

> *Example.* `first` is partial: "`(first empty)`" yields an error.

In FOL, we must use only *total* functions, which have a value (exactly one!) for every possible tuple of arguments.

> *Example.* Interpretation $\mathcal{S}$ cannot interpret a function symbol g as
>
> $$g^{\mathcal{S}} = \{\, \langle \mathtt{x}, \mathtt{(first\ x)} \rangle \mid \mathtt{x} \text{ is a non-empty list} \,\} \quad .$$
>
> This does NOT describe a total function!

(Recall: for any unary g, the formula $\forall x \cdot \exists z \cdot g(x) = z$ is valid.)

Representing Functions That May Be Partial

To describe a Scheme function that is, or may be, partial, we use a relation. For example, `first` corresponds to the relation $\mathcal{R}_{\texttt{first}}$, which we have already seen.

Thus we revise our goal:

> To describe, in FOL, the ternary relation $\mathcal{R}_{\texttt{Append}}$, so that $\mathcal{R}_{\texttt{Append}}(x, y, z)^{(\mathcal{S},\theta)}$ is true if and only if calling `Append` with arguments $\theta(x)$ and $\theta(y)$ produces the result $\theta(z)$.

The Program for Append

The full program for Append:

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

We have many of the parts we need:

Scheme construct	*Logical analogue*
constant or variable (`empty`, `x`, ...)	constant or variable (e, x, ...)
unnamed values	fresh variable (z, ...)
relation (`equal?`, ...)	relation ($=$, ...)
function (`cons`, `first`, ...)	function OR relation ($cons$, $\mathcal{R}_{\texttt{first}}$, ...)

But what to do with cond?

Control Stuctures: `cond`

What does `(cond (a b) C...)` mean?

- If `a` evaluates to `#t`, the `cond` has the same evaluation as `b`.
- If `a` evaluates to `#f`, the `cond` has the same evaluation as `(cond C...)`. ("Remove the `(a b)` and try again.")

If we run out of conditions, getting simply "`(cond)`", then the expression has no meaningful value.

In the case of `Append`, we have `(equal? x empty)` for `a` and `y` for `b`.

Append, line 1

The first line:

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

Corresponding formula:

$$x = e \rightarrow \mathcal{R}_{\texttt{Append}}(x, y, y)$$

Simplified:

$$\mathcal{R}_{\texttt{Append}}(e, y, y) \ .$$

Remaining:

$$x \neq e \rightarrow \varphi_2 \ ,$$

where φ_2 comes from the rest of the cond.

Append, line 2

```
cond ( #t  (cons ( first x )  ( Append (rest x) y ) ) )
```

Introduce new variables for the anonymous values:

v_f: the value of `first x`
v_r: the value of `rest x`
v_a: the value of `(Append ...)`.

The formula φ_2:

$$\mathcal{R}_{\texttt{first}}(x, v_f) \to \Big(\mathcal{R}_{\texttt{rest}}(x, v_r) \to \Big(\mathcal{R}_{\texttt{Append}}(v_r, y, v_a) \to \mathcal{R}_{\texttt{Append}}(x, y, \mathit{cons}(v_f, v_a))\Big)\Big)$$

The formula $x \neq e \to \varphi_2$, simplified (using BL1 and BL2):

$$\mathcal{R}_{\texttt{Append}}(v_r, y, v_a) \to \mathcal{R}_{\texttt{Append}}\big(\mathit{cons}(v_f, v_r), y, \mathit{cons}(v_f, v_a)\big)$$

Formulas describing Append

Thus the following formulas apply to $\mathcal{R}_{\texttt{Append}}$.

App1: $\forall y \cdot \mathcal{R}_{\texttt{Append}}(e, y, y)$

App2: $\forall x \cdot \forall y \cdot \forall z \cdot \forall w \cdot$
$\left(\mathcal{R}_{\texttt{Append}}(x, y, z) \rightarrow \mathcal{R}_{\texttt{Append}}\left(\mathit{cons}\,(w, x)\,, y, \mathit{cons}\,(w, z)\right)\right)$

Exercise:

Prove that Append is total; that is, prove

$$\{\text{App1}, \text{App2}\} \vdash_{\mathit{BList}} \forall y \cdot \forall x \cdot \exists z \cdot \mathcal{R}_{\texttt{Append}}(x, y, z)$$

Properties of `Append`

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.

- Give a formula, using the relation $\mathcal{R}_{\texttt{Append}}$, that states the required property.
- Show that your formula has a proof from the list axioms and App1 and App2.

Predicates and Functions for Basic Lists in FOL

We can add a relation symbol for any Scheme function on lists.

- for a **built-in function**, its definition in Scheme determines the appropriate FOL relation (or function).
- for a **user-defined function**, our goal is to find formulas that characterize the appropriate relation associated to the Scheme function.

But some Scheme functions take arguments that are not lists. . . .

FOL Formulas for Scheme Programs

For FOL formulas for general Scheme programs, we will find it convenient to have objects that are not lists.

To have such objects, we must modify our axioms for basic lists; in particular, the induction scheme BL3 forces every object except e to be a *cons*.

Let $\mathit{atom}(x)$ denote the formula $\forall y \cdot \forall z \cdot x \neq \mathit{cons}(y, z)$.

General lists

To allow both lists and other objects, we must modify the axioms for lists.

Let $atom(x)$ be the formula $\forall y \cdot \forall z \cdot x \neq cons(y, z)$.

GL1: $\forall x \cdot \forall y \cdot cons(x, y) \neq e$.

GL2: $\forall x \cdot \forall y \cdot \forall z \cdot \forall w \cdot cons(x, y) = cons(z, w) \rightarrow (x = z \wedge y = w)$.

GL3: For each formula $\varphi(x)$ and each variable y not free in φ,

$$\forall x \cdot (atom(x) \rightarrow \varphi) \rightarrow \big(\forall x \cdot \big(\varphi \rightarrow \forall y \cdot \varphi[cons(y, x)/x]\big) \rightarrow \forall x \cdot \varphi\big)$$

Only the induction axiom has changed from the basic version.
Its "base case" includes all non-lists in addition to the empty list.

Where Have We Been, and Where Next?

We have seen an example—`Append`—of "translating" a Scheme program into logic.

In principle, one can do the same for any program. (It can, of course, get very complicated. Proving properties from the result can be even more complicated.)

Instead, however, we shall consider some even-more-general issues:

- How can we represent a program itself in logic?
- What does such a representation imply about programs?

Formulas for general Scheme programs

Formulas for Scheme Programs

Goal: To describe, in FOL, how a program executes.

We have to accomplish two main tasks.

1. Represent a program.
2. Describe the execution of a program.

This is much like producing an interpreter or compiler.
But we shall focus on analysis, rather than on actually computing.

Representing Scheme expressions and programs

We shall represent a Scheme program as a list.

Recall the Scheme function `Append`:

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

Let's recall how we treated the parts of this program before....

Remembering "Append"

We analyzed the program for `Append` as follows.

- We introduced variables for the values used in the program, whether named (like `x` and `y`) or unnamed.
- We used FOL constants and relations for the built-in constants, functions and relations (`empty`, `equal?`, `first`, etc.).
- We did not directly represent control structures, such as `cond`. Instead, we used `cond` to determine how to link FOL formulas together.

For a general treatment, we need to handle

- arbitrary names, and
- control structures.

What's in a Name?

For an arbitrary program, we must represent everything uniformly.

We need the concept of a "name" of a value, function, and a way to represent names so that formulas can refer to them.

We need the capability to

- compare names with one another, to determine whether or not they are the same, and
- have a "dictionary" of the meanings of names, so that we can look up a name and replace it with its meaning.

What's in a Name?–II

To do these, we introduce an FOL constant symbol *name*, and adopt the following convention.

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

Thus, we may express the concept, "x is a name," by the formula

$$\exists y \cdot (x = \mathit{cons}(\mathit{name}, y)) \quad .$$

What's in a Name?—III

We assume a canonical way to transcribe text strings into names, and use the notation $\underline{s}$ to mean the name corresponding to the string s.

If s_1 and s_2 are different strings, then the corresponding names $\underline{s_1}$ and $\underline{s_2}$ are also different.

Exceptions: we shall represent the keywords `cond`, `define`, and `lambda` by their own FOL constants, respectively *cond*, *define*, and λ.

Program Append, as a Term of FOL

With these conventions, any Scheme expression can be translated into a term in the language of lists.

Example. The program Append becomes the term

$$\Big\langle \mathit{define}, \langle \underline{\mathit{Append}}, \underline{x}, \underline{y} \rangle, \\ \big\langle \mathit{cond}, \langle \langle \underline{\mathit{equal?}}, \underline{x}, e \rangle, \underline{y} \rangle \big\rangle, \\ \big\langle \underline{\#t}, \langle \langle \underline{\mathit{first}}, \underline{x} \rangle, \langle \underline{\mathit{Append}}, \langle \underline{\mathit{rest}}, \underline{x} \rangle, \underline{y} \rangle \big\rangle \big\rangle \Big\rangle$$

This yields a representation of a program.

What can we do with this representation?

Evaluation of Programs

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, then evaluation has ended.

The "Step" Relation

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,
- a list representing a potential next state.

We want a term $Step(x, D, y)$ to have the value true iff the expression x converts to expression y in one step, given dictionary D.

For this we specify axioms, where each part of the definition of Scheme becomes one or more axiom schemata.

Simplifying Assumptions

For now, we shall make two assumptions about programs.

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

What about a program that uses local variables?

> Answer: simply re-name local variables so that they all have distinct names, and then make them global. This does not affect the execution of the program.

Also, `set!` is not actually needed. (It's really `define` in disguise!)

Various Kinds of Steps

There are several ways to take a step of evaluation.

Simple steps:

- Apply a built-in function of Scheme.
- Replace a user-defined name by its definition.

More-complex steps:

- Apply a control element.
- Apply a user-defined function.
- Take a step in a sub-expression.

If none of the above applies, evaluation stops.

Built-In Functions

If a name denotes a **built-in function** `fun`, then we assume that the function is definable by a FOL relation. That is, there is a formula $\rho_{\texttt{fun}}(x_1, \ldots, x_k, y)$ that is true iff `(fun x1 ... xk)` produces value y.

Example:

The formula $\rho_{\texttt{first}}$ for the built-in function `first` is $\mathcal{R}_{\texttt{first}}(x, y)$.

Applying a Built-In Function

This leads to our first axiom schema for *Step*:

Ax1: $\rho_{\texttt{fun}}(\vec{x}, y) \rightarrow \mathit{Step}(\langle \underline{\mathit{fun}}, \vec{x} \rangle, D, y)$,

for each built-in function `fun`.

Example: Since $\mathcal{R}_{\texttt{first}}(\mathit{cons}(x, y), x)$ holds, Ax1 implies

$$\mathit{Step}(\langle \underline{\mathit{first}}, \mathit{cons}(x, y) \rangle, D, x) \quad .$$

The Dictionary of Other Names

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 FOL, this means that we need a relation *LookUp* such that $LookUp(x, D, y)$ evaluates to true iff dictionary D specifies value y for name x.

Questions:

- What is a dictionary?
- How do we specify the relation *LookUp*?

The Structure of a Dictionary

Abstractly, a *dictionary* is a mapping from names to values.

Concretely, we shall use the standard data structure of an association list—implemented in FOL.

An *association* is simply a pair, whose first element is a name [a.k.a. key, or index] and whose second element is the corresponding value [definition].

An *association list* is a list of associations.

Looking Up a Name, I

If we have a dictionary and a name to look up, there are 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

  Ax2: $\mathit{LookUp}(x, \mathit{cons}(\langle x, y\rangle, z), y)$

- ...

Looking Up a Name, II

- 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

  Ax3: $x \neq u \rightarrow \mathit{LookUp}(x, D, y) \rightarrow \mathit{LookUp}(x, \mathit{cons}(\langle u, v \rangle, D), y)$.

- If the dictionary has no first pair (it is the empty list), then no name has a value in the dictionary.

  (No axiom.)

Stepping Via LookUp

Once we have *LookUp* characterized, we can use it for *Step*.

Ax4: $\big(name(x) \wedge LookUp(x, D, y)\big) \rightarrow Step(x, D, y)$

This completes the simple cases of taking a step.

Next, we discuss the concept of a "value", since we need it for the more-complex steps.

What Is a Value?

Intuitively, a "value" is the result of a computation—i.e., when we obtain a value, we're done computing.

If we start a sub-computation, then when it produces a value, we return the value to the main computation to evaluate further.

In other words, a value is an expression whose evaluation is complete.

Scheme has many kinds of values: numbers, text strings, etc. Also, functions are values.

When we have a list structure, we must look more closely. If `v` and `w` are values, then `(v w)` may or may not be a value. . . .

Lists of Values

Suppose that `v` and `w` are values.

The expression `(v w)` may or may not be a value:

Example 1. If `v` is a number, then `(v w)` is a value—a list of two values.

Example 2. 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.

We need a relation *IsValue*, that allows a distinction.

Defining Relation *IsValue*

- Each Scheme constant c is a value—`empty`, `#t`, etc.
  Also, `(list ...)` is a value.

  Ax5: $\mathit{IsValue}(c)$, for each defined constant c,
  and $\forall y \cdot \mathit{IsValue}(\underline{\mathit{list}}, y)$.

- A `lambda`-expression is a value:

  Ax6: $\forall x \cdot \forall y \cdot \mathit{IsValue}\big(\langle \lambda, x, y \rangle\big)$.

- If x and y are values, and x is not a function, then $\mathit{cons}(x, y)$ is a value:

  Ax7: $\big(\mathit{IsValue}(x) \wedge \mathit{IsValue}(y) \wedge \neg \mathcal{R}_{\texttt{first}}(x, \lambda)\big)$
  $\rightarrow \mathit{IsValue}(\mathit{cons}(x, y))$.

Evaluate the First Expression, If Possible

We now return to evaluation of complex expressions.

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 can be evaluated by itself,**
  then take a step in that expression, leaving the rest unchanged.
  As an axiom, we get

  Ax8: $\mathit{Step}(x, D, y) \rightarrow \mathit{Step}\big(\mathit{cons}(x, z), D, \mathit{cons}(y, z)\big)$.

A Simple "Complex" Expression

Example. The definition

`(define Foo ⟨something⟩ )`

yields the dictionary $\langle\ \langle \underline{Foo}, \underline{\langle something \rangle} \rangle\ \rangle$.

Thus one step of the evaluation of `(Foo a b)` is described by

$$Step\left(\langle \underline{Foo}, \underline{a}, \underline{b} \rangle, \langle\ \langle \underline{Foo}, \underline{\langle something \rangle} \rangle\ \rangle, \langle \underline{\langle something \rangle}, \underline{a}, \underline{b} \rangle\right) \quad .$$

(Hopefully the result will make sense for further steps!)

Control Elements: `cond`

The element `cond` exemplifies control elements.

> **Notation:** We shall write $cons3(x, y, z)$ to mean $cons(x, cons(y, z))$—a list whose first two items are x and y.

Recall the definition of `(cond (a b) c...)`:

- `(cond (#t b) c...)` has the same evaluation as `b`.
  Ax9: $Step\big(cons3(cond, \langle \underline{\#t}, x \rangle, y), D, x\big)$.
- `(cond (#f b) c...)` has the same evaluation as `(cond c...)`:
  Ax10: $Step\big(cons3(cond, \langle \underline{\#f}, x \rangle, y), D, cons(cond, y)\big)$.

cond, Continued

Finally, in `(cond (a b) c...)`, if `a` is not yet evaluated, we must evaluate it first.

Ax11: $Step(z, D, w) \rightarrow$
$Step(cons3(cond, \langle z, x \rangle, y), D, cons3(cond, \langle w, x \rangle, y))$.

Applying Functions

The final case of stepping: evaluating an expression which is the application of a function.

The basic definition in Scheme works by substitution: to apply a function `(lambda (x) y)` to a value `u`, substitute `u` for the name `x` everywhere that `x` occurs in the expression `y`.

If $Subst(\langle x, u\rangle, t, v)$ means "substituting u for x in term y yields term v", then we get

Ax12: $IsValue(u) \rightarrow Subst(\langle x, u\rangle, y, v) \rightarrow Step(\langle\langle \lambda, x, y\rangle, u\rangle, D, v)$.

- Why the "*IsValue*"?
- We require a definition of *Subst*.

Applying Functions, II

The implicant $IsValue(u)$ appeared in the previous axiom because Scheme specifies that functions may only be applied to evaluated arguments.

To handle unevaluated arguments, we simply evaluate them:

Ax13: $Step(u, D, v) \rightarrow Step(\langle\langle\lambda, x, y\rangle, u\rangle, D, \langle\langle\lambda, x, y\rangle, v\rangle)$.

Defining Substitution

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

Ax14: $\mathit{Subst}\,(\langle x, u\rangle, e, e)$ and $\mathit{Subst}\,(\langle x, u\rangle, x, u)$.

and the inductive case

Ax15:

$$\left(\neg\mathcal{R}_{\texttt{first}}(y, \mathit{name}) \wedge \mathit{Subst}\,(\langle x, u\rangle, y, v) \wedge \mathit{Subst}\,(\langle x, u\rangle, z, w)\right) \rightarrow \mathit{Subst}\,(\langle x, u\rangle, \mathit{cons}\,(y, z), \mathit{cons}\,(v, w)) \quad .$$