CORE-verifier Documentation

Release master

Mar 23, 2023

CONTENTS

CORE is a proof verifier for predicate logic. See *Building and Installing* for build and installation instructions.

Note: This project is under active development.

CHAPTER

ONE

BUILDING AND INSTALLING

1.1 Build

With make and gcc installed, clone the official repository first. Navigate into the directory CORE, and run the following:

```
make
```

The binary core is created. A directory build is created containing intermediate object files used during the build process.

1.2 Install

Install CORE manually by copying the binary CORE/core into the appropriate directory (usually /usr/local/bin or /usr/bin).

CHAPTER

TWO

PREDICATE LOGIC

The principal data type in CORE is a first-order sentence.

2.1 Sentences

Sentences in first order intuitionistic logic are the main data type of expressions in CORE. In order to understand how the language works, it's important to understand how to read and interpret these sentences. A *sentence term* is composed of either a *constant*, *predicate*, *relation*, *quantifier*, *negation*, or parentheses term. A sentence term is an example of a sentence. Sentences can be constructed from several sentence terms connected using the binary *logical connectives*.

The `(` character marks the beginning of a parentheses term. The rest of the parentheses term is parsed as the sentence up to a closing `)` character. Parentheses terms may also be nested. Parentheses are used to pick the association of terms in a sentence, which is useful for overriding *Connective Precedence* as in the following example:

```
P(Z) & *X*Y(P(X) & P(Y) -> X = Y) -> *X(P(X) <-> X = Z)
```

Sentences may have any number of *unbound predicates* and *unbound objects*. These sentences have conditional truth values; the truth of the sentence depends on what propositions or objects are substituted.

2.2 Objects

An object is an symbol which may be referenced by *predicates* or *relations* in a *sentence*. Objects may be quantified, or created by the programmer. Objects cannot be overwritten if they are referenced in any *sentence expression value*.

An object identifier must begin with a letter or an underscore, and may subsequently contain letters, digits, and underscores. By convention, object identifiers use only upper-case letters. Objects share the same namespace as *variables*.

In the default library of proofs, objects represent sets in ZFC set theory. However, objects may represent anything the axioms describe.

Note: Objects are only labels. No internal information about an object is stored. Instead, *Sentences* may describe objects.

2.2.1 Creation

Objects may be created locally by *quantifiers*. Unquantified objects are created with either the *given* command or the *pipe* operator applied to a sentence beginning with an *existential* quantifier.

2.2.2 Examples

The following are examples of *valid* object identifiers:

- `NATURALS`
- `X`
- `_temp0`

The following are examples of *invalid* object identifiers:

- `0_TEST`
- `A<A`
- `~OBJ~`

2.3 Logical Constants

There are two logical constants which can be used to construct *sentences*: `true` and `false`. These constants are case sensitive.

2.3.1 Examples

The following are examples of well formed sentences which use logical constants:

```
true
true -> true | false
~^X(false)
```

2.4 Predicates

A predicate is a symbol which may reference *objects* in a *sentence*. A predicate may depend on zero or more arguments (objects), and may or may not have a *sentence definition*. Predicates are a convenient tool for formally defining a concept and for hiding complicated sentences under a label.

A *sentence definition* is a sentence used to define a predicate. If a predicate depends on `n` arguments, the sentence definition has `n` unbound variables. A predicate applied to `n` objects is a sentence logically equivalent to substituting in order those `n` objects for the unbound variables in the sentence definition.

A predicate identifier must begin with either a letter or an underscore, and may subsequently contain letters, digits, and underscores.

Predicates are quantified only by *axiom schema* and *proof schema*.

2.4.1 Creation

Bound predicates are created in the scope of a *proof schema*. Bound predicates have no sentence definition.

Predicates can also be created in any scope using the define command.

2.4.2 Examples

Suppose `in` is a *relation* representing set membership and `subset` is a predicate defined by the following sentence:

$$\forall X \forall Y (\text{subset}(X, Y) \leftrightarrow \forall Z (Z \in X \to Z \in Y)))$$

In CORE's syntax, this sentence would be written as:

```
*X*Y(subset(X, Y) <-> *Z(Z in X -> Z in Y))
```

Then the sentence

```
*X(subset(X, X))
```

would mean:

```
*X*Z(Z in X -> Z in X)
```

2.5 Relations

A relation is a symbol which may reference exactly two *objects* in a *sentence*. Functionally, relations serve the same role as *predicates*. The advantage of a relation is that it provides a more convenient notation for concepts such as equality, set membership, and subsets.

Relations may or may not have a *sentence definition*. If a relation does have a sentence definition, the sentence has two unbound variables sharing the order in which the objects would appear when related by the relation.

2.5.1 Syntax

A relation's identifier may appear between two object identifiers in a sentence:

```
[object identifier] [relation identifier] [object identifier]
```

A sentence formed like the one above creates a *sentence term*.

Relation identifiers may either begin with a letter, underscore, or a character which is not one of `;`, `(`, or `)`. If a relation begins with a letter or underscore, the subsequent characters of a relation identifier must be either letters, underscores, or digits. Otherwise, the subsequent characters must be any character which is not a letter, underscore, whitespace, `;`, `(`, or `)`.

2.5.2 Creation

Relations may be created in any scope using the relation command.

2.5.3 Examples

The following are valid relation identifiers:

- `in`
- `subset`
- `=`
- `<=`
- `<69`

While the following are invalid relation identifiers:

- `hi<=`
- `0=`
- `< =`

Here are some example sentence terms and the corresponding object identifiers and relation identifiers:

- `A<B` object identifiers: `A` and `B`, relation identifier: `<`
- `X_SET in C` object identifiers: `X_SET` and `C`, relation identifier: `in`

2.6 Connectives

Logical connectives create new *sentences* from two or more constituent sentences. There are five logical connectives in CORE: `~`, `&`, `|`, `->`, and `<->`.

2.6.1 Negation

The logical connective `~` represents negation. Unlike the other four connectives, this is a unary connective. If `A` represents a well formed sentence, then `~A` is a well formed sentence. The sentence `~A` should be interpreted to mean that `A` is not true.

Sentences composed with the `~` connective can be proven using the *not* command. A sentence of the form `~A` is logically equivalent to the sentence `A -> false`. Given `A` and `~A`, one can *deduce* `false`.

Note: Since CORE uses a kind of intuitionistic logic, it is not true in general that `A | ~A` for any sentence of the form `A`. Additionally, `~~A` is generally weaker than the sentence `A`. One can model classical logic by adding the law of excluded middle as an *axiom*.

2.6.2 And

The logical connective & represents "and." If `{A}` and `{B}` represent well formed sentences, then `{A} & {B}` is a well formed sentence. `{A} & {B}` is interpreted to be true if and only if `{A}` is true and `{B}` is true.

Sentences composed with the & logical connective can be constructed using the built-in function and. A sentence composed with the & connective can be deconstructed into its constituent sentences using variable assignment. A sentence `{C}` is stronger than the sentence `{A} & {B}` if `{C}` is stronger than both `{A}` and `{B}`. The sentence `{A} & {B}` is stronger than a sentence `{C}` if either `{A}` or `{B}` is stonger than `{C}`. See Trivial Implication for more information.

2.6.3 Or

The logical connective | represents "or." If `{A}` and `{B}` represent well formed sentences, then `{A} | {B}` is a well formed sentence. `{A} | {B}` is interpreted to be true if and only if `{A}` is true or `{B}` is true.

Sentences composed with the | logical connective can be constructed using the built-in function or. A sentence composed with the | connective can be used in a proof to branch an argument. A sentence `{C}` is stronger than the sentence `{A} | {B}` if `{C}` is stronger than either `{A}` or `{B}`. The sentence `{A} | {B}` is stronger than a sentence `{C}` if both `{A}` and `{B}` are stronger than `{C}`. See Trivial Implication for more information.

2.6.4 Implies

The logical connective -> represents implication. If `{A}` and `{B}` represent well formed sentences, then `{A} -> {B}` is a well formed sentence. `{A} -> {B}` is interpreted to be true if and only if whenever one can prove `{A}`, one can prove `{B}`.

Verified sentences `{A} -> {B}` and `{A}` can be used to *deduce* the verified sentence `{B}`.

2.6.5 Biconditional

The logical connective <-> represents the biconditonal connective. If `{A}` and `{B}` represent well formed sentences, then `{A} <-> {B}` is a well formed sentences. `{A} <-> {B}` is logically equivalent to `({A} -> {B}) & ({B} -> {A})`.

Sentences composed with the <-> logical connective can be constructed from implications using the built-in function iff. Verified sentences `{A} <-> {B}` and `{A}` can be used to *deduce* the verified sentence `{B}`. In addition, verified sentences `{A} <-> {B}` and `{B}` can be used to *deduce* `{A}`.

2.6.6 Connective Precedence

Connectives are associated according to precedence. Connectives are in the following list in order from highest precedence to lowest precedence:

1. &
2. |
3. ->
4. <->

For example, the sentence

```
A | B & C <-> D -> E
```

Is parsed as:

```
(A | (B & C)) <-> (D -> E)
```

Due to connective precedence.

2.7 Quantifiers

Quantifiers bind new *objects* in *sentences*. CORE has two quantifiers: universal and existential. Objects bound by a quantifier mask existing objects with the same identifier in the following sentence term.

2.7.1 Syntax

```
*[object identifier] [sentence term]
^[object identifier] [sentence term]
```

The universal and existential quantifiers are denoted by `*` and `^` respectively. After a quantifier follows a *sentence term* which may refer to the variable bound by the quantifier.

2.7.2 Universal Quantifiers

Universal quantifiers are denoted by `*`. If `P` is a sentence with an unbound *object*, then the sentence `*X(P(X))` is interpreted to mean that for all objects `X`, the sentence `P(X)` is true.

A sentence of the form `*X(P(X))` can be proven using the *given* command. A proven sentence of the form `*X(P(X))` can be used to deduce `P(Y)` for any object with identifier `Y` using *object substitution*.

2.7.3 Existential Quantifiers

Existential quantifiers are denoted by `^`. If `P` is a sentence with an unbound *object*, then the sentence `^X(P(X))` is interpreted to mean that there exists an object `Y` such that the sentence `P(Y)` is true.

A sentence of the form `^X(P(X))` can be proven using the *choose* command. A proven sentence of the form `^X(P(X))` can be used to construct an object `Y` in the current scope such that `P(Y)` is true. See *Pipe Operator* for more information.

2.7.4 Examples

If = is a relation and `subset` is a predicate depending on two objects, then one can form the following sentence:

```
*X*Y subset(X, Y) & subset(Y, X) -> X = Y
```

If the objects `X` and `Y` do not exist in the current scope or any parent scope, then this sentence is semantically invalid. The reason is that `subset(X, Y)` is a *sentence term*, so the sentence is parsed as:

```
((*X*Y(subset(X, Y))) & subset(Y, X)) -> (X = Y)
```

In this example, the objects `X` and `Y` which are quantified are only accessible within the *sentence term* `subset(X, Y)`. If `X` and `Y` are objects accessible in the current scope, however, then this is a valid sentence. To avoid ambiguity, it is best practice to add parentheses after a sequence of quantifiers, as in the following example:

```
*X*Y(subset(X, Y) & subset(Y, X) -> X = Y)
```

2.8 Trivial Truth

In order to balance convenience for the programmer with precision, CORE has a notion of when a sentence is *trivially true*.

2.8.1 Trivially True

Suppose `P` is a sentence. To determine whether `P` is trivially true, search for the first applicable rule from the beginning in the following list:

1. If `P` is of the form `A -> B` for sentences `A` and `B`, `P` is trivially true if `A` *trivially implies* `B`.
2. If `P` is of the form `A <-> B`, `P` is trivially true if `A` trivially implies `B` and `B` trivially implies `A`
3. If `P` is of the form `A & B`, `P` is trivially true if `A` is trivially true and `B` is trivially true.
4. If `P` is of the form `A | B`, `P` is trivially true if `A` is trivially true or `B` is trivially true.
5. If `P` is of the form `*X(A(X))`, `P` is trivially true if `A` is trivially true.
6. If `P` is of the form `~A`, `P` is trivially true if `A` is trivially false.
7. If `P` is the sentence `true`, `P` is trivially true.

One should interpret the trivial truth of a sentence with or without unbound variables to mean that no matter which objects are substituted for the unbound variables, the sentence can easily be prove true. A case for the trivial truth of a sentence beginning with an *existential quantifier* does not exist because not all models may have objects.

2.8.2 Trivially False

Suppose `P` is a sentence. To determine whether `P` is trivially false, search for the first applicable rule from the beginning in the following list:

1. If `P` is of the form `A -> B` for sentences `A` and `B`, `P` is trivially false if `A` is trivially true and `B` is trivially false.
2. If `P` is of the form `A <-> B`, `P` is trivially false if either `A -> B` or `B -> A` is trivially false.
3. If `P` is of the form `A & B`, `P` is trivially false if either `A` is trivially false or `B` is trivially false.
4. If `P` is of the form `A | B`, `P` is trivially false if `A` is trivially false and `B` is trivially false.
5. If `P` is of the form `^X(A(X))` or the form `*X(A(X))`, `P` is trivially false if `A` is trivially false.
6. If `P` is of the form `~A`, `P` is trivially false if `A` is trivially true.
7. If `P` is the sentence `false`, `P` is trivially false.

One should interpret the trivial falsity of a sentence with or without unbound variables to mean that no matter which objects are substituted for the unbound variables, the sentence can easily be proven false.

2.8.3 Trivial Implication

Suppose `P` and `Q` are sentences. To determine whether `P` *trivially implies* `Q`, search for the first applicable rule from the beginning of the following list. At most one rule is applied, so if a rule is applied and the rule does not give trivial implication, then `P` does not trivially imply `Q`.

1. If `P` and `Q` have mismatching numbers of unbound variables or predicates, then `P` does not trivially imply `Q`.
2. If `P` is of the form `A | B`, `P` trivially implies `Q` if `A` trivially implies `Q` and `B` trivially implies `Q`.
3. If `Q` is of the form `A & B`, `P` trivially implies `Q` if `P` trivially implies `A` and `P` trivially implies `B`.
4. If `P` is trivially false, then `P` trivially implies `Q`.
5. If `Q` is trivially true, then `P` trivially implies `Q`.
6. If `P` is of the form `~A` and `Q` is of the form `~B`, `P` trivially implies `Q` if `B` trivially implies `A`.
7. If `Q` is of the form `~~A`, `P` trivially implies `Q` if `P` trivially implies `A`.
8. If `P` is of the form `A -> B` and `Q` is of the form `C -> D`, then `P` trivially implies `Q` if `C` trivially implies `A` and `B` trivially implies `D`.
9. If `P` is of the form `*X(A(X))` and `Q` is of the form `*X(B(X)`, then `P` trivially implies `Q` if `A` trivially implies `B`.
10. If `P` is of the form `^X(A(X))` and `Q` is of the form `^X(B(X))`, then `P` trivially implies `Q` if `A` trivially implies `B`.
11. **If `P` is of the form `A <-> B` and `Q` is of the form `C <-> D`, then apply any of the following rules:**
    - If `A` trivially implies `C`, `C` trivially implies `A`, `B` trivially implies `D`, and `D` trivially implies `B`, then `P` trivially implies `Q`.
    - If `A` trivially implies `D`, `D` trivially implies `A`, `B` trivially implies `C`, and `C` trivially implies `B`, then `P` trivially implies `Q`.
    - If `C` trivially implies `D` and `D` trivially implies `C`, then `P` trivially implies `Q`.
12. If `P` and `Q` are identical *relation sentence terms*, then `P` trivially implies `Q`.
13. If `P` and `Q` are identical *predicate sentence terms*, then `P` trivially implies `Q`.
14. **Otherwise, apply any of the following rules:**
    - If `P` is of the form `A & B` and either `A` trivially implies `Q` or `B` trivially implies `Q`, then `P` trivially implies `Q`.
    - If `Q` is of the form `A | B` and either `P` trivially implies `A` or `P` trivially implies `B`, then `P` trivially implies `Q`.

Regardless of the number of unbound variables of `A` and `B`, one should interpret "`A` trivially implies `B`" to mean that no matter which objects are substituted for the unbound variables, one can easily prove that `A` implies `B`.

2.8.4 Trivial Equivalence

Suppose `P` and `Q` are sentences. To determine whether `P` is *trivially equivalent* to `Q`, determine whether `P` trivially implies `Q` and `Q` trivially implies `P`. If so, then `P` is trivially equivalent to `Q`, otherwise `P` is not trivially equivalent to `Q`.

CHAPTER

THREE

EXPRESSIONS

Expressions allow the programmer to combine expression values to obtain new expression values. Expression operations include deduction rules.

3.1 Variables

CORE has two types of variables: sentence variables and context variables. Variables share the same namespace as *objects*, so a variable and an object can't share the same name in the same scope.

3.1.1 Sentence Variables

Sentence variables store either verified or unverified sentence expression values. Expressions which refer to a sentence variable by name evaluate to that variable's stored sentence value. For example, if `result` is a sentence variable, then `result` is an expression evaluating to the sentence value stored by the variable `result`. Sentence variables stored in the current scope can be overwritten at any point in the current scope. When the current scope is destroyed, the sentence variables stored in the scope are destroyed with it.

Variable identifiers may begin with either a letter or an underscore, and may subsequently contain letters, digits, and underscores.

3.1.2 Context Variables

Context variables store the contents of a scope created with the context command. Sentence variables, *objects*, *definitions*, and relations stored inside of a context scope can be referenced inside of sentence literals and expressions using the `.` operator. For example, if `con` is a context and `name` is a result, object, definition, or relation inside of the context, then `con.name` refers to it. Context variables can store scopes with more context variables. For instance, if `con` stores a context variable `con2`, and `con2` stores `name`, then this variable can be referenced as `con.con2.name`.

Context variables cannot be overwritten. The scopes stored by a context variable are immutable.

3.2 Object Substitution

Objects accessible from the current scope can be substituted for an object bound by a *universal quantifier* using *object substitution*. Object substitution results in a new sentence expression value without the universal quantifier.

3.2.1 Syntax

```
[sentence expression]([object expression], [object expression], ...)
```

Object substitution begins with a *parent expression* evaluating to a *sentence value*. The sentence referenced by the sentence value must begin with a top level *universal quantifier*. In other words, the sentence must be of the form `*X(P(X))` for some sentence `P` with one unbound object. Following the parent expression is `(` and one or more expressions evaluating to *object expression values*. The `(` is matched with a closing `)`.

For each object expression value, a top-level universal quantifier is peeled from the parent expression sentence, and the unbound variable is replaced with the object.

Object substitution can only be applied when the parent expression evaluates to a sentence value with no unbound variables or predicates. If the parent expression evaluates to a *verified sentence value*, then the object substitution evaluates to a verified sentence value. Otherwise, the object substitution evaluates to an *unverified sentence value*. In any case, the new sentence value has no bound variables or predicates.

3.2.2 Examples

If = and <= are *relations*, `S` and `T` are *objects*, `subset_equal` is a *sentence variable* storing the following sentence:

```
*X*Y(X <= Y & Y <= X -> X = Y)
```

Then the expression:

```
subset_equal(S, T)
```

Evaluates to the sentence value:

```
S <= T & T <= S -> S = T
```

If the sentence stored by `subset_equal` is verified, then this sentence value is also verified, otherwise it is unverified.

3.3 Pipe Operator

New objects can be constructed from a verified sentence value beginning with a top-level *existential quantifier* using the *pipe operator*.

3.3.1 Syntax

```
[sentence expression]|[object identifier], [object identifier], ...|
```

A pipe operator sentence term begins with a *parent expression* evaluating to a *verified sentence value*. The sentence referenced by the sentence value must begin with a top level *existential quantifier*. In other words, the sentence must be of the form `^X(P(X))` for some sentence `P` with one unbound object. Following the parent expression is the character `|`, one or more *object identifiers*, and a final `|` character.

For each object identifier, a top level existential quantifier is peeled from the parent expression sentence. A new object in the current scope is created with the object identifier, overwriting any variables or objects sharing the same identifier (This raises an error if the object or variable cannot be overwritten). Then, this new object is substituted for the unbound variable in the sentence. After doing this for each object identifier, the pipe operator sentence term evaluates to the remaining sentence.

The pipe operator can only be applied to a parent expression evaluating to a *verified sentence value* with no unbound predicates. Consequently, a pipe operator sentence term always evaluates to a verified sentence value with no unbound predicates.

3.3.2 Examples

If `in` is a *relation*, `S` and `T` are *objects*, and `axiom_pairing` is a *sentence variable* storing the following verified sentence:

```
*X*Y^Z(X in Z & Y in Z)
```

Then the expression `axiom_pairing(S, T)|P|` creates an object `P` and evaluates to the following verified sentence:

```
S in P & T in P
```

Working in set theory where `in` means set membership, `P` is now a set which contains both `S` and `T`. This new sentence value can act as a definition for `P`.

3.4 Deduction

Deduction is applied to *sentence values* containing a top level *implication* to prove a conclusion given a premise.

3.4.1 Syntax

```
[expression value]([expression value], [expression value], ...)
```

A deduction sentence term begins with a *parent expression* evaluating to a *sentence value*. Following the parent expression is the character `(`, one or more argument expressions evaluating to sentence values, and the character `)`. The parent expression sentence value must begin with a top level *implies*, *biconditional*, or *not* connective.

Each of the argument sentence values are linked together using the *and* connective. The resulting sentence will be referred to as the *premise sentence*

If the parent expression sentence value is *verified* and every argument sentence value is verified, then the deduction sentence term evaluates to a verified sentence value. Otherwise, it evaluates to an *unverified sentence value*.

3.4.2 Implication Deduction

If the parent expression sentence is of the form `A -> B` for sentences `A` and `B`, then it is checked that the premise sentence *trivially implies* the sentence `A`. If not, an error is raised. The deduction sentence term evaluates to the sentence value `B`.

3.4.3 Biconditional Deduction

If the parent expression sentence is of the form `A <-> B` for sentences `A` and `B`, then it is checked that the premise sentence either *trivially implies* `A` or `B`. If not, an error is raised. If the premise sentence implies `A`, then the deduction sentence term evaluates to the sentence `B`, otherwise it evaluates to the sentence value `A`.

3.4.4 Negation Deduction

If the parent expression sentence is of the form `~A` for the sentence `A`, then it is checked that the premise sentence *trivially implies* `A`. If not, an error is raised. The deduction sentence term evaluates to the sentence `false`.

3.4.5 Examples

Suppose `in` is a relation; `A`, `B`, `C`, and `D` are objects; and `union_D` is a sentence variable storing the following sentence value:

```
A in B & B in C -> A in D
```

If `A_in_B` is a sentence variable storing `A in B` and `B_in_C` is a sentence variable storing `B in C`, then the expression `union_D(A_in_B, B_in_C)` evaluates to the sentence value `A in D`.

Suppose `equal` is a sentence variable storing the following sentence value:

```
A in C | B in C -> A = B
```

If `B_in_C` is a sentence variable storing `B in C`, then the expression `equal(B_in_C)` evaluates to the sentence value `A = B`.

3.5 Sentence Constants

Sentence constants provide a way for the programmer to produce any *unverified sentence value* without proof. Sentence constants are most often used for predicate substitution of *axiom schema* or *proof schema*.

3.5.1 Syntax

```
<[predicate identifier]([number of arguments]), ..., [object identifier], ... :␣
↪[sentence]>
```

A sentence constant expression term begins with the character <. What follows is a list of labels for unbound predicates and unbound objects. After this list is the character `:`, and a sentence depending on the identifiers in the preceding list. The sentence constant expression is ended with the character >. The list of unbound predicate and object labels may optionally be empty, in which case the `:` character directly follows the < character.

An unbound predicate in the list of identifiers is denoted using a *predicate identifier*, parentheses, and the number of arguments. If the number of arguments is zero, then the number of arguments may be omitted; however, in this case the parentheses must still be present, otherwise the identifier will be added as an unbound object instead of an unbound predicate.

An unbound object in the list of identifiers is denoted using an *object identifier*.

Unbound objects and predicates may be listed in any order, but the n-th unbound predicate of the output sentence value is the n-th listed unbound predicate. Likewise, the n-th unbound object of the output sentence value is the n-th listed unbound object.

The output is an *unverified sentence value* storing the given sentence with the given unbound predicates and unbound objects.

3.5.2 Examples

If `in` is a *relation*, then the following expression:

```
<A, B: A in B | B in A>
```

Evaluates to an unverified sentence with two unbound objects `A` and `B` storing the sentence `A in B | B in A`.

The following expression:

```
<P(1): ^X(P(X) & ~P(X))>
```

Evaluates to an unverified sentence with no unbound objects and one unbound predicate `P(1)` storing the sentence `^X(P(X) & ~P(X))`. It does not matter that the sentence is clearly not true since the output of the expression is an *unverified* sentence value.

3.6 Predicate Definitions

Predicate definitions can be used to obtain verified sentences. If a *predicate* or *relation* has a sentence definition, then by definition the predicate or relation applied to objects makes a true sentence when the corresponding sentence definition with those objects substituted is true.

3.6.1 Syntax

```
#[identifier]
```

A predicate definition expression term begins with a `#` character followed by an identifier. The identifier may be either a *predicate identifier* or a *relation identifier*.

3.6.2 Behavior

There must be a predicate or relation which is accessible from the current *scope* and shares the same identifier, otherwise an error is raised. If the predicate or relation does not have a sentence definition, then an error is raised.

If a predicate `P` depending on n objects shares the same identifier, denote the sentence definition as `A(X1, X2, ..., Xn)` for unbound objects `X1`, `X2`,. . . , `Xn`. The output of the expression is a verified sentence of the following form:

```
*X1*X2*[...]*Xn(A(X1, X2,..., Xn) <-> P(X1, X2,..., Xn))
```

If a relation `R` shares the same identifier, denote the sentence definition as `A(X1, X2)` for unbound objects `X1` and `X2`. The output of the expression is a verified sentence of the following form:

```
*X1*X2(A(X1, X2) <-> X1 R X2)
```

3.6.3 Examples

Suppose `in` is a relation and `=` is a relation such that for unbound objects `A` and `B`, `A = B` if and only if:

```
*X(X in A <-> X in B)
```

In other words, the above sentence is the sentence definition for `=` where `A` and `B` are the first and second unbound objects.

Suppose now that `X`, `A`, and `B` are objects, `eq` is a variable storing the following verified sentence:

```
A = B
```

And `member` is a variable storing the following verified sentence:

```
X in A
```

Then the following expression:

```
#=(A, B)(eq)(X)(member)
```

Evaluates to the verified sentence:

```
X in B
```

If `successor` is a predicate depending on two objects such that for unbound objects `C` and `D`, `successor(C, D)` if and only if:

```
*Y(Y in C <-> Y in D | Y = D)
```

Then the following expression:

```
#successor
```

Evaluates to the following verified sentence:

```
*C*D(*Y(Y in C <-> Y in D | Y = D) <-> successor(C, D))
```

3.7 Expression Brackets

One must often apply *axiom schema* and *proof schema* to specific predicates. This is accomplished using expression brackets. Expression brackets allow the substitution of *predicates* for unbound predicates in verified and unverified sentences.

3.7.1 Syntax

```
[parent expression][[predicate expression], [predicate expression], ...]
```

An expression brackets term begins with a parent expression followed by a `[` character, a comma-separated list of predicate expressions, and a closing `]` character.

The parent expression value must evaluate to a sentence value with at least one unbound *predicate*. Each predicate expression must evaluate to a sentence value. The n-th predicate sentence is substituted for the n-th unbound predicate in the parent sentence, and the expression brackets term evaluates to the result. The n-th predicate sentence must have the same number of unbound objects as the number of arguments of the n-th unbound predicate of the parent sentence.

The expression brackets term evaluates to a verified sentence if and only if the parent sentence is verified.

3.7.2 Examples

Suppose `excluded_middle` is a variable storing the following verified sentence for unbound predicate `P()`:

```
P | ~P
```

Then we can use expression brackets to apply this result to any predicate depending on no arguments. For example, suppose `X` and `Y` are objects, and < is a *relation*. Then the following expression:

```
excluded_middle[<: X < Y>]
```

Evaluates to the following verified sentence:

```
X < Y | ~X < Y
```

Suppose that = is a relation and that `replace` is a variable storing the following verified sentence for unbound predicate `P(1)`:

```
*X*Y(X = Y -> (P(X) <-> P(Y)))
```

If `Z`, `W`, and `V` are objects and < is a *relation*, then the following expression:

```
replace[<Q: Q < V>](Z, W)
```

Evaluates to the following verified sentence:

```
Z = W -> (Z < V <-> W < V)
```

Note that *object substitution* is applied to the result of the expression brackets term.

3.8 Built-in Functions

There are a few built-in functions for manipulating *expression values* in CORE. Each built-in function expression begins with an identifier followed by a `(` character. A built-in function expression term evaluates to the output of the function. You can determine the behavior of the built-in functions in the documents below.

3.8.1 left

The built-in function `left` accepts one *sentence expression value* and returns the left-hand side of the sentence. It's precise behavior depends on the input value.

Syntax

```
left([sentence argument expression])
```

Behavior

If the argument does not evaluate to a sentence expression value or the argument sentence has unbound objects or propositions, an error is raised.

The following lists the behavior of the function depending on the argument sentence type:

- If the argument sentence is of the form `A & B`, then the output is the sentence `A`. The output is verified if and only if the argument sentence is verified.
- If the argument sentence is of the form `A | B`, then the output is the sentence `A`. The output is unverified.
- If the argument sentence is of the form `A -> B`, then the output is the sentence `A`. The output is unverified.
- If the argument sentence is of the form `A <-> B`, then the output is the sentence `B -> A`. The output is verified if and only if the argument sentence is verified.
- If the argument sentence is not any of these types, an error is raised

Note: Since this function's output depends on the association of the operations composing the argument sentence which normally doesn't matter, this function's use is not recommended. Oftentimes this function's purpose is over-shadowed by *trivial implication* and variable assignment.

Examples

If `X` and `Y` are objects, `<` is a relation, the output of the following expression:

```
left(<: X < Y & Y < X>)
```

Is the unverified sentence:

```
X < Y
```

If `X` is an object, `empty` is a *definition*, `in` is a *relation*, and `test` stores the verified sentence:

```
^E(empty(E) & E in X) & ~X in X
```

Then the output of the following expression:

```
left(test)
```

Is the verified sentence:

```
^E(empty(E) & E in X)
```

3.8.2 right

The built-in function `right` accepts one *sentence expression value* and returns the right-hand side of the sentence. It's precise behavior depends on the input value.

Syntax

```
right([sentence argument expression])
```

Behavior

If the argument does not evaluate to a sentence expression value or the argument sentence has unbound objects or propositions, an error is raised.

The following lists the behavior of the function depending on the argument sentence type:

- If the argument sentence is of the form `A & B`, then the output is the sentence `B`. The output is verified if and only if the argument sentence is verified.
- If the argument sentence is of the form `A | B`, then the output is the sentence `B`. The output is unverified.
- If the argument sentence is of the form `A -> B`, then the output is the sentence `B`. The output is unverified.
- If the argument sentence is of the form `A <-> B`, then the output is the sentence `A -> B`. The output is verified if and only if the argument sentence is verified.
- If the argument sentence is not any of these types, an error is raised

Note: Since this function's output depends on the association of the operations composing the argument sentence which normally doesn't matter, this function's use is not recommended. Oftentimes this function's purpose is overshadowed by *trivial implication* and variable assignment.

Examples

If `X` and `Y` are objects, < is a relation, the output of the following expression:

```
right(<: X < Y & Y < X>)
```

Is the unverified sentence:

```
Y < X
```

If `X` is an object, `empty` is a *definition*, `in` is a *relation*, and `test` stores the verified sentence:

```
^E(empty(E) & E in X) & ~X in X
```

Then the output of the following expression:

```
right(test)
```

Is the verified sentence:

```
~X in X
```

3.8.3 and

The built-in function `and` accepts one or more *sentence expression values* and returns a sentence which links each argument sentence using the `&` *connective*.

Syntax

```
and([sentence argument expression], ...)
```

Behavior

`and` accepts one or more argument expressions. If any argument expression does not evaluate to a *sentence expression value* or any argument sentence has unbound objects or predicates, then an error is raised. Otherwise, the output is a sentence composed of each argument sentence joined in order using the `&` *connective*.

The output sentence value is verified if every argument sentence value is verified.

Examples

If `in` is a *relation*, `arg0` stores the verified sentence:

```
*X*Y(~X in Y | ~Y in X)
```

And `arg1` stores the verified sentence:

```
~^X(X in X)
```

Then the following expression:

```
and(arg0, arg1)
```

Evaluates to the following verified sentence:

```
*X*Y(~X in Y | ~Y in X) & ~^X(X in X)
```

Additionally, the following expression:

```
and(arg0, <:^X^Y(X in Y & Y in X)>, <:^E*X(~X in E)>)
```

Evaluates to the following unverified sentence:

```
*X*Y(~X in Y | Y in X) & ^X^Y(X in Y & Y in X) & ^E*X(~X in E)
```

3.8.4 or

The built-in function `or` accepts one or more *sentence expression values* and returns a sentence which links each argument sentence using the `|` *connective*.

Syntax

```
or([sentence argument expression], ...)
```

Behavior

`or` accepts one or more argument expressions. If any argument expression does not evaluate to a *sentence expression value* or any argument sentence has unbound objects or predicates, then an error is raised. Otherwise, the output is a sentence composed of each argument sentence joined in order using the `|` *connective*.

The output sentence value is verified if at least one argument sentence value is verified.

Examples

If `in` is a *relation*, `arg0` stores the verified sentence:

```
*X*Y(~X in Y | ~Y in X)
```

And `arg1` stores the verified sentence:

```
~^X(X in X)
```

Then the following expression:

```
or(arg0, arg1)
```

Evaluates to the following verified sentence:

```
*X*Y(~X in Y | ~Y in X) | ~^X(X in X)
```

Additionally, the following expression:

```
or(arg0, <:^X^Y(X in Y & Y in X)>, <:^E*X(~X in E)>)
```

Evaluates to the following verified sentence:

```
*X*Y(~X in Y | Y in X) | ^X^Y(X in Y & Y in X) | ^E*X(~X in E)
```

3.8.5 iff

The built-in function `iff` accepts two *sentence expression values*, each of which are converse implications, and returns the a sentence with a top-level *biconditional connective*.

Syntax

```
iff([sentence argument expression], [sentence argument expression])
```

Behavior

Each argument must evaluate to a *sentence expression value* with no unbound objects or predicates. Both arguments must evaluate to a sentence of the form `A -> B` for some sentences `A` and `B`, otherwise an error is raised. If the first argument is of the form `A -> B` and the second argument is of the form `C -> D`, the verifier checks that `A` is trivially equivalent to `D` and that `B` is trivially equivalent to `C`. If not, an error is raised. The output is the sentence `A <-> B`.

The output sentence expression value is verified if both arguments are verified.

Examples

If `A`, `B`, and `C` are objects and `in` is a relation, then the output of the following expression:

```
iff(<: A in C -> B in C>, <: B in C -> A in C>)
```

Is the unverified sentence value:

```
A in C <-> B in C
```

If `arg0` is a variable storing the verified sentence value `B in C -> A in C` and `arg1` is a variable storing the verified sentence value `A in C -> B in C`, then the expression:

```
iff(arg0, arg1)
```

Returns the verified sentence value:

```
B in C <-> A in C
```

3.8.6 trivial

The built-in function `trivial` accepts one *sentence expression value*, and if the sentence is *trivially true*, returns the same sentence value except verified.

Syntax

```
trivial([sentence argument expression])
```

Behavior

The argument must be an expression evaluating to a *sentence expression value* with no unbound objects or predicates. If the argument sentence is not *trivially true*, then an error is raised. Otherwise, the output is a verified sentence value storing the same sentence as the argument.

Note: Anything that can be proven using the `trivial` built-in command can be proven without it. `trivial` is usually used to save the programmer the time of having to prove simple statements that are obviously true. Due to the limited scope of what is defined as trivially true, there are other sentences that are just as obvious but must be proven manually.

Examples

If `in` is a relation, the following expression:

```
trivial(<: *X*Y(X in Y -> ~~X in Y)>)
```

Returns the following verified sentence value:

```
*X*Y(X in Y -> ~~X in Y)
```

3.8.7 expand

The built-in function `expand` accepts one *sentence expression value* and uses the predicate or relation's corresponding *sentence definition* to expand the sentence.

Syntax

```
expand([sentence argument expression])
```

Behavior

The argument must be an expression evaluating to a *sentence expression value*. If the argument is not a top-level *predicate* or *relation*, then an error is raised. If the argument has any unbound objects or predicates, an error is raised. If the predicate or relation has no sentence definition, then an error is raised.

If the argument is a top-level predicate, then the return value is a copy of the sentence definition with substituted unbound objects. The n-th unbound object of the sentence definition is substituted with the n-th argument of the predicate. The resulting sentence is returned.

If the argument is a top-level relation, then the return value is a copy of the sentence definition with the two unbound objects substituted. The first unbound object of the sentence definition is substituted with the first object related. The second unbound object of the sentence definition is substituted with the second object related. The resulting sentence is returned.

The returned sentence value is verified if and only if the argument sentence value is verified.

Examples

Suppose in is a relation with no sentence definition and = is defined so that for objects X and Y, X = Y if and only if the following sentence is true:

```
*Z(Z in X <-> Z in Y)
```

In other words, the above is the sentence definition of = when viewing X and Y as the first and second unbound objects.

Now suppose that NAT and W are two objects, and the variable equal stores the verified sentence:

```
NAT = W
```

Then the following expression:

```
expand(equal)
```

Evaluates to the following verified sentence:

```
*Z(Z in NAT <-> Z in W)
```

Now suppose successor is a predicate defined so that for any two objects X and Y, successor(X, Y) if and only if:

```
*Z(Z in X <-> Z in Y | Z = Y)
```

In other words, the above is the sentence definition of successor when regarding X and Y as the first and second unbound objects.

If A and B are objects, then the following expression:

```
expand(<: successor(A, B)>)
```

Evaluates to the unverified sentence:

```
*Z(Z in A <-> Z in B | Z = B)
```

3.8.8 substitute

The built-in function substitute allows for the replacement of a sub-sentence with a *biconditionally equivalent* sub-sentence.

Syntax

```
substitute([sentence argument expression])
```

Behavior

The argument must be an expression evaluating to a *sentence expression value*. The argument may or may not be verified, but it must have no unbound objects and exactly one unbound predicate, otherwise an error is raised.

The output expression value is a verified sentence with no unbound objects and two unbound predicates. Let the argument sentence be denoted by `A` with unbound predicate `P` depending on n objects. Let `Q` be a new unbound predicate depending on `n` objects. Let `B` denote the sentence in which each occurence of `P` in `A` is replaced with `Q`. Then the output sentence is the sentence of the following form:

```
*X1*X2*[...]*Xn(P(X1, X2, [...], Xn) <-> Q(X1, X2, [...], Xn)) -> (A <-> B)
```

The first `[...]` means that there are n *quantifiers*, and the second `[...]` means that each of those n quantifiers are the arguments of `P` and `Q`.

Examples

Intuitively, the substitute command allows us to conclude that `P` can be replaced with `Q` if `P` and `Q` mean the same thing.

For example, suppose = is a relation, and `symmetry` is a variable storing the following verified sentence:

```
*X*Y(X = Y <-> Y = X)
```

Furthermore, suppose `transitivity` is a variable storing the following verified sentence:

```
*X*Y*Z(X = Y & Y = Z -> X = Z)
```

Then the following expression:

```
substitute(<P(2): *X*Y*Z(X = Y & P(Y, Z) -> P(X, Z))>)[<A, B: A = B>, <A, B: B = A>
↪](symmetry)(transitivity)
```

Evaluates to the following verified sentence:

```
*X*Y*Z(X = Y & Z = Y -> Z = X)
```

3.8.9 branch

The built-in function `branch` returns a sentence of the form `C` if `A | B` is true, `C` can be proven from `A`, and `C` can be proven from `B`.

Syntax

```
branch([sentence argument expression], [variable identifier], [variable identifier], ...)
↪{
        ...
        return [return expression];
} or {
        ...
        return [return expression];
} or ...
}
```

Behavior

`branch` accepts one *sentence expression value* and two or more *variable identifiers*. The argument sentence must be verified, have a top-level `|` *connective*, and not have an unbound object or predicate, otherwise an error is raised.

After the `)` character is one or more *scopes*. Between consecutive scopes is `or`. The number of scopes must be the same as the number of variable identifiers.

In the n-th scope, a variable with the n-th variable identifier is created.

Inside of the first scope, the argument sentence up to the first top-level `|` is stored as a verified sentence for the first variable identifier. At the end of the scope, a verified sentence must be returned. The return value of `branch` is the return value of the first scope.

For n > 1, inside of the n-th scope the argument sentence between the (n-1)-th top-level `|` and the n-th top-level `|` (or the end of the sentence if the scope is the last scope) is stored as a verified sentence for the n-th variable identifier. At the end of the scope, a verified sentence must be returned. If the return value of the scope is not *trivially equivalent* to the return value of the first scope, an error is raised.

Examples

Suppose = and `in` are relations, and that `successor` is a predicate such that `successor(A, B)` if and only if:

```
*X(X in A <-> X in B | X = B)
```

In other words, suppose the above sentence is the sentence definition of `successor` when `A` and `B` are considered unbound objects.

Suppose `X`, `A`, and `B` are objects, `result` is a predicate depending on one object, `lem0` stores the verified sentence:

```
*Y(Y in B -> result(B))
```

And `lem1` stores the verified sentence:

```
X = B -> result(B)
```

Finally, suppose `arg0` stores the verified sentence:

```
successor(A, B)
```

And `arg1` stores the verified sentence:

```
X in A
```

Then the following expression:

```
branch(expand(arg0)(X)(arg1), X_in_B, X_eq_B){
        return lem0(X)(X_in_B);
} or {
        return lem1(X_eq_B);
}
```

Evaluates to the verified sentence:

```
result(B)
```

3.9 Expression Values

Expressions evaluate to expression values. There are three types of expression values: *verified sentences*, *unverified sentences*, and *object values*.

Verified sentences represent results that the programmer has proven to be true. Verified sentences can generally only be *deduced* from other verified sentences. Verified sentences may have unbound predicates (see *Axiom Schema* and *Proof Schema*), but will never have unbound variables.

Unverified sentences are generally substituted as a predicate in a sentence with unbound predicates. An expression value containing any unverified sentence can be created using *sentence constants*.

Object expression values are used for *object substitution*. These expression values only store a reference to an object. The object refered by an object expression value will never be an object bound by a quantifier in a sentence. An object expression value is created by referring to the object by name. For example, if an object named `OBJ` belongs to a context called `con`, then the expression `con.OBJ` evaluates to an object expression value referencing that object.

CHAPTER

FOUR

COMMANDS

Commands in CORE run inside of *scopes*. Commands may change the state of the program by, for example, creating *definitions*, creating relations, *assigning variables*, and *proving* theorems.

4.1 Scopes

Scopes are environments in which commands run. Scopes are used to separate the program's states. For example, *variable assignment* in one scope can't overwrite variables inside of another scope.

4.1.1 Syntax

```
{
        [one or more commands]
}
```

```
{
        [one or more commands]
        return [expression];
}
```

Each scope begins with a `{` character and ends with a `}` character. Inside each scope is one or more commands, most of which must be terminated with a `;` character. Optionally, a scope may end with a return command.

4.1.2 Behavior

Each scope creates a new *namespace* for variables, objects, definitions, and relations.

Scopes can be created inside of scopes, in which case the new scope becomes a *child scope* of the current scope. The *parent scopes* of the child scope include the current scope and any of the current scope's parent scopes. A variable, object, definition, or relation is accessible inside of a scope if and only if that entity is in the current namespace or any parent scope's namespace.

All variables, objects, definitions, and relations in a scopes namespace are destroyed at the end of the scope except in the case in which the scope was created by the context command.

Global scopes are the top-level scopes created at the beginning of program execution. Dependencies can be added within a global scope.

Dependent scopes are child scopes created by the `dependent context` command. Besides global scopes, dependent scopes are the only scopes in which dependencies can be added.

4.2 print

The `print` command allows the programmer to display an *expression value* in the console. The command is often used for debugging while developing a proof.

4.2.1 Syntax

```
print [expression];
```

The `print` command begins with the keyword `print` followed by a `;` terminated expression.

4.2.2 Behavior

If `[expression]` evaluates to an *object value*, then the following is printed in the console:

```
'[file name]' line [line number]: [object identifier]
```

Where `[object identifier]` is the identifier of the object referred to by the object value.

If `[expression]` evaluates to a *verified sentence value*, then the following is printed in the console:

```
'[file name]' line [line number]: (verified)   [sentence]
```

Where `[sentence]` is the sentence referred to by the sentence value.

If `[expression]` evaluates to a *unverified sentence value*, then the following is printed in the console:

```
'[file name]' line [line number]: (unverified) [sentence]
```

Where `[sentence]` is the sentence referred to by the sentence value.

When a sentence is printed by the `print` command, the sentence is printed in *normal form*. In normal form, each quantified or unbound object is replaced with numbers 0, 1, 2, etc. The nth unbound object is always renamed to the object n in the printed sentence value. Then the remaining quantified variables' identifiers are replaced with the next available number.

Additionally, unbound predicates in normal form are denoted in order by `P0`, `P1`, `P2`, etc.

4.2.3 Examples

The following command:

```
print trivial(<: true>);
```

Prints the following line in the console:

```
'[file name]' line [line number]: (verified)   true
```

The following command:

```
print <Q(3), Z: *X*Y(Q(X, Y, Z) <-> Q(Z, X, Y))>
```

Prints the following line in the console:

```
'[file name]' line [line number]: (unverified) *1*2(P0(1, 2, 0) <-> P0(0, 1, 2))
```

4.3 object

The `object` command is used to assert that an object exists.

4.3.1 Syntax

```
object [object identifier];
object [object identifier], [object identifier], ...;
```

The object identifier begins with the keyword `object` followed by one or more comma-separated *object identifiers*. The `object` command is terminated with a `;` character.

4.3.2 Behavior

For each object identifier, an *object* is created in the current scope with the corresponding object identifier. For each such object, an object dependency is created in the current scope.

Because the `object` command creates dependencies, the `object` command may only be used in a *global scope* or a *dependent scope*.

4.3.3 Examples

In the default proofs library, the program `naturals.core` creates two objects `NATURALS` and `ZERO` using the commands:

```
object NATURALS;
object ZERO;
```

Which can be condensed into the single command:

```
object NATURALS, ZERO;
```

4.4 axiom

The `axiom` command is used to assert the truth of a sentence and store the sentence in a *sentence variable* as a *verified sentence value*.

4.4.1 Syntax

```
axiom [axiom identifier]: [sentence];
```

The axiom command begins with the keyword `axiom`, followed by a variable identifier. A `:` character denotes the beginning of a `;` terminated sentence.

4.4.2 Axiom Schema

```
axiom [axiom identifier][[predicate0]([num args]), [predicate1]([num args]), ...]:␣
↪[sentence];
```

Optionally, after the axiom identifier a list of unbound predicates may follow. The list of unbound predicates begins with a `[` character. This is followed by a comma-separated list of predicates. Each entry in the list begins with a predicate identifier. After the predicate identifier, the number of arguments of the predicate depends on is denoted with parentheses and

4.4.3 Behavior

The sentence specified in the axiom command, which may depend on the optional list of unbound predicates, is stored as a *verified sentence value* into the *variable* with the identifier `[axiom identifier]`.

For each axiom, an axiom dependency is created in the current scope. Since the `axiom` command creates a dependency, the command can only be used in a *global scope* or a *dependent scope*.

Axioms with unbound predicates act as axiom schema through the use of *expression brackets*.

4.4.4 Examples

Suppose < is a relation. Then the following commands:

```
object ZERO;
axiom zero_minimal: *X(~X < ZERO);
```

Create an object named `ZERO` and an axiom which asserts that no object `X` is related to `ZERO` by <, or with the usual interpretation of <, no object is less than `ZERO`.

The following command:

```
axiom strong_induction[P(1)]: *N(*M(M < N -> P(M)) -> P(N)) -> *N(P(N));
```

Encodes second-order strong induction as an axiom schema. The axiom can be applied to any predicate depending on one argument.

4.5 define

The `define` command creates a new definition in the current *scope* and a *predicate* which may be referred to by *sentences* in the current or children scopes.

4.5.1 Syntax

```
define [definition name]: [sentence definition];
define [definition name]([arg0], [arg1], ...): [sentence definition];
define [definition name];
define [definition name]([arg0], [arg1], ...);
dependent define [definition name];
dependent define [definition name]([arg0], [arg1], ...);
dependent define [definition name]([arg0], [arg1], ...): [sentence definition];
```

The command may optionally begin with the `dependent` keyword before the `define` keyword. The following identifier must be a valid *predicate identifier*. An optional list of *object identifiers* may follow the definition name. The `;` character may terminate the command, or `:` marks the start of a `;` terminated sentence.

4.5.2 Behavior

The list of object identifiers is a list of unbound objects in the sentence definition, if present. The number of arguments the predicate depends on is the number of identifiers in the list of object identifiers. If no list of object identifiers is given, the predicate created depends on no arguments. If `[sentence definition]` is present, then it is the sentence definition of the predicate created, otherwise the predicate has no sentence definition.

If the `dependent` keyword is present or `[sentence definition]` is not present, then the definition is added as a definition dependency. In these cases, the command must be called in a *global scope* or a *dependent scope*.

4.5.3 Examples

Suppose `in` is a *relation*. Then consider the following commands:

```
object NATURALS;
define successor(A, B): *X(X in A -> X in B | X = B);
axiom successor_natural: *N*M(M in NATURALS & successor(N, M) -> N in NATURALS);
```

An object `NATURALS` is created as well as a definition `successor`. The sentence `successor(A, B)` should be interpreted as "`A` is the successor of `B`." Then the axiom `successor_natural` says that the successor of a member of `NATURALS` is a member of `NATURALS`.

Now consider the following command:

```
dependent define related(A, B);
```

The command creates a definition `related` which has two arguments but no sentence definition. When used in a program which is imported, it *asserts* that a definition with the identifier `related` exists in the importing program, but does not enforce any requirements for such a definition. Such a command is useful when one wants to assert that a definition which behaves in a certain way (specified by *axioms*) exists but does not care how the definition is constructed.

4.6 Variable Assignment

Variable assignment stores one or more *sentence values* in *sentence variables*. Sentences connected using the *and* connective can be split and stored using variable assignment.

4.6.1 Syntax

```
[identifier0], [identifier1], ... = [expression];
```

Variable assignment begins with a comma-separated list of one or more variable identifiers. This is followed with the = character and an expression. The command is terminated using the `;` character.

4.6.2 Behavior

If there are `n` comma-separated variable identifiers, `[expression]` must evaluate to a *sentence value* which is `n` or more sentences connected with a top-level `&` connective, otherwise an error is raised.

Starting from the first identifier, the left-most sentence in `[expression]` connected using a top-level `&` is peeled from the sentence and stored in the identifier. This process is repeated for the new sentence and each of the following identifiers. The remainder of the sentence is stored for the last variable.

4.6.3 Examples

In the following example, an object `NAT` is created and an axiom which claims that no object is `in` itself. This axiom is applied using *object substitution* with `NAT`, and the resulting sentence is *stored* in `result` for later use.

```
relation in;
object NAT;
axiom recursive_member: *X(~X in X);
result = recursive_member(NAT);
```

The following example demonstrates how variable assignment can be used to unpack two or more sentences which are important on their own:

```
relation in;
object A, B;
axiom pair_member: *X*Y(~X in Y & ~Y in X);
A_not_in_B, B_not_in_A = pair_member(A, B);
```

After execution, `A_not_in_B` stores the sentence `~A in B`, and `B_not_in_A` stores the sentence `~B in A`.

4.7 prove

The `prove` command verifies a sentence and stores it as a *verified sentence value* into a *sentence variable*.

4.7.1 Syntax

```
prove [identifier]: [sentence]{
        ...
}
prove [identifier][[predicate0](num_args), [predicate1], ...]: [sentence]{
        ...
}
```

The `prove` command begins with the keyword `prove` followed by a variable identifier. Optionally, a list of predicate identifiers may be present. The list begins with the `[` character, and ends with the `]` character. Each entry in the list is a predicate identifier, followed by an optional number of arguments. If present, the number of arguments is denoted by the `(` character, a non-negative integer, and the `)` character. Entries in the predicate list are separated with the `,` character.

Following the variable identifier and optional predicate identifiers is the `:` character and a sentence. The command is terminated with a new *scope*.

4.7.2 Behavior

The sentence `[sentence]` may depend on the bound *predicates* specified in the predicate list. If the code in the scope successfully executes, this sentence is stored as a verified *sentence value* into the *sentence variable* `[identifier]`.

The execution of the code inside of the new scope depends on sentence specified by `[sentence]`. This sentence is called the *goal* of the scope.

The following commands only run in a scope with a *goal*:

assume

The `assume` command creates a new scope with a modified goal. It is used when the goal in the current scope is an implication. The `assume` command represents the action of "assuming the premises" in a proof.

Syntax

```
assume [identifier0], [identifier1], ...;
assume [identifier0], [identifier1], ...{
        ...
}
```

The `assume` command begins with the keyword `assume`, followed by a comma-separated list of variable identifiers. The command is terminated with either the `;` character or a new scope.

Behavior

When the assume command is used, the current goal must have a top-level implication, otherwise an error is raised. Suppose the goal is of the form `A -> B` for sentences A and B. A new scope is created, and the contents of A are unpacked into the variables with identifiers given by the identifier list in the new scope. In the new scope, the goal is the sentence B.

Starting from the first identifier, the left-most sentence in A connected using a top-level & is peeled from the sentence and stored in the identifier. This process is repeated for the new sentence and each of the following identifiers. The remainder of the sentence is stored for the last variable. This unpacking process is the same as that for *assignment*.

If the commands in the new scope are verified successfully and the goal is *proven*, then the goal is proven in the parent scope. No commands can follow the assume command in the parent scope.

If assume command is terminated with a ; character, the new scope and parent scope end with the next unmatched } character. Otherwise, only the new scope ends with the next unmatched } character.

Examples

Using assume with the return command, some simple proofs can be showcased. The following examples are valid:

```
prove implies_transitive[P, Q, R]: (P -> Q) & (Q -> R) -> (P -> R){
        assume P_implies_Q, Q_implies_R;
        assume P_true;
        return Q_implies_R(P_implies_Q(P_true));
}
```

After execution, the sentence `(P -> Q) & (Q -> R) -> (P -> R)` depending on unbound predicates P, Q, and R is stored in the variable `implies transitive` in the example above.

```
relation in;
object Z;
object Y;
object X;
axiom transitive_Z: *V*W(V in Z & W in V -> W in Z);

prove X_in_Z_if: X in Y & Y in Z -> X in Z{
        assume X_in_Y, Y_in_Z;
        return transitive_Z(Y, X)(X_in_Y, Y_in_Z);
}
```

After execution, the sentence `X in Z` is stored in the variable `X_in_Z` in the example above.

not

The not command creates a new scope with a modified goal. It is used when the goal in the current scope is a top-level negation. The not command represents the action of "assuming the premise" and "proving a contradiction" in a proof.

If the current goal is of the form ~A for a sentence A, using the not command operates as if the *assume* command was used with a goal of `A -> false`.

Syntax

```
not [identifier];
not [identifier]{
        ...
}
```

The not command begins with the keyword not, followed by a variable identifier. The command is terminated with either the ; character or a new scope.

Behavior

When the not command is used, the current goal must have a top-level negation, otherwise an error is raised. Suppose the goal is of the form ~A for the sentence A. A new scope is created, and the contents of A are stored into the variable with identifier [identifier]. In the new scope, the goal is the sentence false. Proving false in the new scope represents finding a contradiction.

If the commands in the new scope are verified successfully and the goal is proven, then the goal is proven in the parent scope. No commands can follow the not command in the parent scope.

If not command is terminated with a ; character, the new scope and parent scope end with the next unmatched } character. Otherwise, only the new scope ends with the next unmatched } character.

Examples

The following examples showcase the not command:

```
prove demorgan0[P, Q]: ~P | ~Q -> ~(P & Q){
        assume one_false;
        not P_and_Q;
        return branch(one_false, not_P, not_Q){
                return not_P(P_and_Q);
        } or {
                return not_Q(P_and_Q);
        };
}

prove demorgan1[P, Q]: ~P & ~Q -> ~(P | Q){
        assume not_P, not_Q;
        not P_or_Q;
        return branch(P_or_Q, P_true, Q_true){
                return not_P(P_true);
        } or {
                return not_Q(Q_true);
        };
}

prove demorgan2[P, Q]: ~(P | Q) -> ~P & ~Q{
        assume not_either;
        prove not_P: ~P{
                not P_true;
                return not_either(P_true);
```

(continues on next page)

(continued from previous page)

```
        }
        prove not_Q: ~Q{
                not Q_true;
                return not_either(Q_true);
        }
        return not_P, not_Q;
}
```

Note: The remaining version of De Morgan's law (`~(P & Q) -> ~P | ~Q`) cannot be proven without assuming the law of excluded middle.

given

WIP

choose

WIP