FORMULA 2.0: A Language for Formal Specifications

Ethan K. Jackson and Wolfram Schulte

Microsoft Research, Redmond, WA
{ejackson,schulte}@microsoft.com

Abstract. FORMULA 2.0 is a novel formal specification language based on *open-world logic programs* and *behavioral types*. Its goals are (1) succinct specifications of domain-specific abstractions and compilers, (2) efficient reasoning and compilation of input programs, (3) diverse synthesis and fast verification. We take a unique approach towards achieving these goals: Specifications are written as strongly-typed open-world logic programs. They are highly declarative and easily express rich synthesis / verification problems. Automated reasoning is enabled by efficient symbolic execution of logic programs into constraints. This tutorial introduces the FORMULA 2.0 language and concepts through a series of small examples.

1 Data and Types

1.1 Constants

FORMULA specifications define, examine, translate and generate *data*. The simplest kinds of data are *constants*, which have no internal structure. Every FORMULA specification has access to some predefined constants. *Numeric constants* can be (arbitrarily) long integers:

```
-1098245634534545630234, 0, 1, 2, 3098098445645649034
```

Or, they can be pairs of integers separated by a '.', i.e. decimal fractions:

```
-223423.23422342342, 0.0, 1.5, 10.8798723000000000000000003
```

Or, they can be pairs of integers separated by a '/', i.e. fractions:

```
-223423/23422342342, 4/8, 9873957/987395487987334
```

FORMULA converts numerics into normalized fractions; no precision is lost. For example, the following equalities are true:

```
2/3 = 6/9, 1/2 = 0.5, 1.0000 = 1
```

The following disequalities are true:

```
2/3 != 0.66667, 0 != 0.000000000000000000000000000001
```

Operations on numerics do not lose precision. Infinities are not explicitly part of FORMULA's vocabulary. For example, the fraction '1/0' causes a syntax error.

ASCII strings are also supported. One way to write a string is by enclosing it in double-quotes. These strings must end on the same line where they began, so we refer to them as *single-line strings*. Here are some examples:

```
"", "Hello World", "Foo\nBar"
```

Sometimes it is necessary to put special characters inside of a string. This can be accomplished using the C-style escape character '\'. Table 1 gives the escape sequences.

Some strings are unpleasant to escape, such as strings containing code or filenames with backslashes. Multi-line strings capture all the text within a pair of delimiters, including line breaks. A multi-line string starts with a single-double-quote pair `'` `"` and ends with a double-single-quote pair `"` `'`. Below is an example; note '\' is not an escape character in multi-line strings:

```
'" "This\string has funny 'thi\ngs in it'" "'
```

The only sequence that needs to be escaped in a multi-line string is the sequence terminating the string. For symmetry, the starting sequence also has an escape code (see Table 2). For example, the following equality is true:

```
'" ''"" ""'' \"' = " '\" \"' \\"
```

The final kind of constant is the *user-defined constant*. Syntactically, user-defined constants are identifiers. Here are some examples of user-defined constants:

```
TRUE, FALSE, RED, GREEN, NIL
```

Note that `TRUE` and `FALSE` are automatically defined on the user's behalf, though they are not keywords.

By convention, the names of user-defined constants should consist of all uppercase characters and be at least two characters long. This convention helps to distinguish constants from other identifiers. Two user-defined constants denote the same value are if and only if they have the same. For example:

```
TRUE = TRUE, FALSE = FALSE, TRUE != FALSE, RED != BLUE
```

1.2 Data Constructors

Complex data values are created by functions called *data constructors* (or *constructors* for short). An n-ary data constructor f takes n data values as arguments and returns a new data value. Here are some examples:

```
Person("John", "Smith"),
Node(1, Node(2, NIL, NIL), Node(3, NIL, NIL))
```

The $Person(,)$ constructor creates *Person* values from two string arguments. The $Node(,,)$ constructor builds binary trees of integer keys. The arguments to $Node$ are: (1) an integer key, (2) a left-subtree (or the constant *NIL* if none) and (3) a right-subtree (or the constant *NIL* if none).

Data constructors are proper functions, i.e. they always produce the same values from the same inputs. Contrast this with the following object-oriented program:

```
class Node {
  ...
  Node (int Key, Node left, Node right)
  { ... }
}

Node x = new Node(1, null, null);
Node y = new Node(1, null, null);
if (x != y) {
   print("Different");
}
```

Table 1. Table of single-line string escapes

Single-Line String Escapes	
Syntax	Result
`\n`	Produces a line feed.
`\r`	Produces a carriage return.
`\t`	Produces a tab.
`\`x	Produces x for $x \notin \{\mathtt{n}, \mathtt{r}, \mathtt{t}\}$, e.g. `\\` or `\"` .

Table 2. Table of multi-line string escapes

Multi-Line String Escapes	
Syntax	Result
`''""`	Produces the sequence `'"`.
`""''`	Produces the sequence `"'`.

The program prints the string "Different" because x and y hold different nodes, even though the nodes were constructed with the same values. In FORMULA two values are the same if and only if (1) they are the same constant, or (2) they were constructed by the same constructor using the same arguments. For example:

```
NIL = NIL, NIL != FALSE,
Node(1, NIL, NIL) = Node(1, NIL, NIL),
Person("John", "Smith") != Node(2, NIL, NIL)
```

As this example shows, values can be compared using the *equality* '`=`' and *disequality* '`!=`' relations. These relations are defined for arbitrary pairs of values.

1.3 Ordering of Values

Values are also ordered. The ordering relation on values is called a *lexicographic order*, which generalizes the dictionary order of strings. First, we split all values into four ordered families: numerics, strings, user constants, and *complex values*:

Definition 11 (Ordering of Families)**.**

$$\mathit{family}(x) \overset{def}{=} \begin{cases} 0 & \text{if } x \text{ is a numeric,} \\ 1 & \text{if } x \text{ is a string,} \\ 2 & \text{if } x \text{ is a user constant,} \\ 3 & \text{otherwise.} \end{cases}$$

Families yield a *precedence relation* $\ll$ on values:

$$x \ll y \text{ if } \mathit{family}(x) < \mathit{family}(y).$$

Definition 12 (Ordering of Values)**.** For values x and y, define $x < y$ if $x \neq y$ and any of the following are satisfied:

- $x \ll y$.
- Both values are numerics; x comes before y on the real number line.
- Both values are strings; x comes before y in dictionary order (assuming a case-sensitive order using the ASCII encoding of characters).
- Both values are user constants; the name of x comes before the name of y in dictionary order.
- Both values are complex, i.e. $x = f(t_1, \ldots, t_n)$ and $y = g(s_1, \ldots, s_m)$, and any of the following are satisfied:
  - The name of constructor f comes before the name of the constructor g in dictionary order.
  - Both constructors have the same name, and the first i where $t_i \neq s_i$ then $t_i < s_i$.

Here are some examples:

```
0 < 1, 1 < "FALSE", "FALSE" < FALSE, FALSE < TRUE
```

Table 3. Table of built-in data types

Built-in Data Types	
Name	Meaning
`Real`	The set of all numeric values.
`Integer`	The set of all integers.
`Natural`	The set of all non-negative integers.
`PosInteger`	The set of all positive integers.
`NegInteger`	The set of all negative integers.
`String`	The set of all string integers.
`Boolean`	The set of constants `TRUE` and `FALSE`.

```
Node(1, Node(10, NIL, NIL), NIL) < Node(2, NIL, NIL),
    Node(2, NIL, NIL) < Person("John", "Smith")
```

The predefined relations <, <=, >, and >= use this order. The predefined functions `min` and `max` find the smallest and largest values according to <. Finally, all predefined functions that must sort values, e.g. `toList`, also use this order.

1.4 Data Types and Subtyping

A *data type* (or just a *type*) is a expression standing for a set of values. Table 3 lists the built-in data types and their meanings. In addition, other types can be defined. Suppose $f(...)$ is an n-ary constructor, then the type f stands for the range of the constructor f. Suppose c is a constant, then the type $\{c\}$ stands for the singleton set containing c. Suppose τ_1 and τ_2 are types, then $\tau_1 + \tau_2$ stands for the set-union of the two types. Finally, $f(\tau_1, \ldots, \tau_n)$ stands for the set of all values obtained by applying f to all possible values in $\tau_1, \ldots, \tau_n$. FORMULA provides some special syntax to make it easier to write types. A *finite enumeration* is a set of constants:

```
{ RED, GREEN, FALSE, 1, 2, "Hello" }
```

An enumeration can also include integer ranges:

```
{ -1000..1000, 1001..1001, 1002 }
```

This type stands for the set of all strings, integers, and Boolean values:

```
Real + String + { TRUE, FALSE }
```

This type stands for the set of all integer-keyed binary trees with a non-empty left child:

```
Node(Integer, Node, Node + { NIL })
```

Also, this type stands for the set of all integer-keyed binary trees with a non-empty right child:

```
Node(Integer, Node + { NIL }, Node)
```

Data types are related to each other by the *subtyping relation*. In object-oriented languages subtyping is indicated by explicitly subclassing a base class, extending an interface, or implementing an interface. In FORMULA the subtyping relationship is determined implicitly by the values a type represents. A type τ_1 is a subtype of τ_2 if the values represented by τ_1 are a *subset* of those represented by τ_2. We write τ_1 `<:` τ_2 if τ_1 is a subtype of τ_2. Here are some examples of types satisfying the subtyping relationship:

```
{ 1, 2 } <: PosInteger <: Natural + String <: Real + String
```

```
Node(Integer, Node, Node)<: Node(Integer, Node, Node + {NIL})
```

```
Node(Integer, Node, Node + {NIL}) <: Node
```

The above examples show that types are fairly precise; they can represent very specific sets of data. Also, types are *behavioral*; FORMULA only cares about the set of values a type stands for, but not how the type is written. For example, all of these types mean the same:

```
{ 0..10 } + Natural = Natural = {0} + PosInteger
```

```
Node(Integer, Node + {NIL}, Node + {NIL}) = Node
```

FORMULA infers types for expressions, and these types over-approximate the evaluation of expressions. We write $e : \tau$ if the expression e is assigned the type τ. For instance, a *C++* compiler might assign the *int* type to the expression 1 or the $float$ type to the expression 1.5. Because FORMULA types are more precise, the type of a value is just a singleton set containing that value, i.e. $1 : \{1\}$ and $1.5 : \{\frac{3}{2}\}$. Because subtyping is implicit by subset inclusion, the value 1 can be used anywhere a *Real* is accepted without coercion. This is unlike *C++*, where the integer value 1 would be coerced to the floating point value 1.0 because *int* and $float$ are different kinds of values.

Consider the more complicated *C++* example:

```
enum E { Zero = 0, One = 1, Two = 2 };
bool Foo(E x, E y)
{
   auto z = x + y;
   return z > 10;
}
```

The *C++* compiler infers $z : int$, even though the expected values for x and y are in the interval $[0, 2]$. Also, z is always less than 10 and $z > 10$ is always false. Consider an analogous FORMULA specification:

```
1: transform Foo(x: E, y: E) returns (b:: Bool)
2: {
3:    E ::= { 0..2 }.
4:    Return(TRUE) :- z = %x + %y, z > 10.
5: }
```

The *transform* takes two parameters x and y of type E (defined in line 3). The rule in line 4 is triggered whenever $z = x + y$ and $z > 10$. FORMULA infers that $x, y : \{0..2\}$, $z : \{0..4\}$, $z > 10 : \{\texttt{FALSE}\}$. Finally, it issues an error because the condition $z > 10$ can never be satisfied. (We explain the structure of rules in the next section.) Thus, type inference can be used to catch errors in specifications.

1.5 Type Declarations

Type declarations are used to: (1) define simple names for complicated type expressions, (2) introduce new data constructors along with the types of their arguments, (3) introduce new user-defined constants. Type declarations come in two forms. The first form assigns a name to a type expression.

```
TypeName ::= TypeExpr.
```

The second form introduces a new data constructor.

```
ConstructorName ::= (Arg1_TypeExpr, ..., ArgN_TypeExpr).
```

The expressions appearing in declarations are restricted; they cannot contain constructor applications such as *Node*(*Integer*, *NIL*, *NIL*). However, it is legal to use the type *Node*, which stands for the entire range of the *Node*(, ,) constructor. This restriction allows for more efficient type inference and type manipulation.

Type declarations must be placed in modules, which are self-contained units. The meaning of a type declaration is understood w.r.t. all the type declarations within the same module. In the examples to follow we use *domain modules* to hold type declarations. For now it is enough to understand that type declarations are not visible outside of their modules. (We describe domains in detail in the next section.) Below are two modules D and D' that define the type *Id* in different ways:

```
domain D { Id ::= Integer. } domain D' { Id ::= String. }
```

In domain D the type *Id* stands for integers and in D' it stands for strings. The FORMULA compiler accepts these declarations because they occur in two distinct modules. On the other hand these declarations are illegal.

Error: Conflicting definitions of type Id

```
domain D
{
  Id ::= Integer.
  Id ::= String.
}
```

There are multiple conflicting declarations of the type *Id* in the same module. FORMULA accepts any set of type declarations as long as their *meaning* is consistent across the module. For example, this module is legal because it defines *Id* in two equivalent ways.

Legal: All definitions of type Id are equivalent

```
domain D
{
  Id ::= Integer.
  Id ::= NegInteger + {0} + PosInteger.
}
```

1.6 Declaring Constants

User-defined constants are implicitly declared by using them in some enumeration in some type declaration. Every domain automatically contains the declaration:

```
Boolean ::= { TRUE, FALSE }.
```

Introduces user-defined constants RED, GREEN, and BLUE.

```
domain Colors
{
  NamedColor ::= { RED, GREEN, BLUE }.
  Color      ::= { RED, GREEN, BLUE, 0..16777215 }.
}
```

The *NamedColor* type contains the constants *RED*, *GREEN* and *BLUE*. These constants are implicitly declared by using them in the type declaration. The *Color* type also mentions these constants along with all integers in the 24-bit RGB color spectrum. User-defined constants are only distinguished by name, so every occurrence of *RED* stands for the same user-defined constant. Unlike *C++*, a user-defined constant is not equivalent to an integer.

In C++ user-defined constants are actually integers; not possible in FORMULA.

```
 enum Color { RED = 0xFF0000, GREEN = 0x00FF00, BLUE = 0x0000FF };
```

Unlike *C#*, user-defined constants exist independently of the type declaration in which they are introduced.

```
 C# introduces constants NamedColors.RED, NamedColors.GREEN,
NamedColors.BLUE
  enum NamedColor { RED, GREEN, BLUE }
 C# introduces constants Colors.RED, Colors.BLUE, Colors.GREEN
  enum Color { RED, GREEN, BLUE }
```

1.7 Declaring Data Constructors

Data constructors are declared by special syntax. The left-hand side of the declaration is the name of the constructor and the right-hand side is a comma-separated list of argument types with parenthesis. Every constructor must have at least one argument, otherwise it would be constant. Here is another domain providing constructors for colors:

```
1: domain Colors
2: {
3:   NamedColor ::= (String).
4:   RGBColor   ::= (r: {0..255}, g: {0..255}, b: {0..255}).
5:   RGBAColor  ::= (a: {0..255}, r: {0..255}, g: {0..255}, b: {0..255}).
6:   Color      ::= NamedColor + RGBColor + RGBAColor.
7: }
```

NamedColor (line 3) defines a unary constructor taking a string. This constructor can be used to create values such as:

```
NamedColor("RED"), NamedColor("GREEN"), NamedColor("BLUE")
```

It is illegal to apply the *NamedColor* color constructor to values other than strings. Also the corresponding *NamedColor* type is automatically defined and only contains those values that obey argument types. The *RGBColor* constructor (line 4) takes three arguments for the red, green, and blue components. The arguments have been given explicit names r, g, and b. Naming arguments is optional but useful. Finally, the *Color* type is a union of the possible color values. As before, there are no implicit conversions between values. For instance:

```
RGBColor(0, 0, 255) != 255 != RGBAColor(0, 0, 0, 255).
```

Every constructor creates distinct values.

Data constructors are similar to *structs* or records in *C*-like languages, but more general. Consider the task of defining a node struct in *C#*. The following code is illegal because the struct *Node* directly depends on *Node* values.

Cannot define a struct that depends directly on itself.

```
struct Node
{
   int key; Node left; Node right;
   Node(int k, Node l, Node r)
   { key = k; left = l; right = r; }
};
```

The problem is that *Node* must have a default value, and there is no way to construct this default value. The definition becomes legal if *struct* is replaced with *class* and then *null* is a valid default value for the *left* and *right* fields. However, this comes with the price that node equality is significantly weakened, i.e. $n == m$ only if the variables n and m hold the same reference. In other words, binary trees cannot be compared as if they were just values.

The declarations of FORMULA constructors can cyclically depend on themselves. Here is the equivalent definition for a *Node* in FORMULA.

```
domain Trees
{
   Node ::= (key  : Integer,
             left : Node + {NIL},
             right: Node + {NIL}).
}
```

The only requirement on constructors is that there must be some arguments of finite size satisfying the type constraints of the constructor. For example, a minimal node value can be constructed by:

```
Node(0, NIL, NIL)
```

Note that *NIL* is not a keyword; just a user-defined constant. This specification has an error because there is no way to construct a node value using a finite number of applications.

```
domain Trees
{
   Node ::= (key  : Integer,
             left : Node,
             right: Node + {NIL}).
}
```

The problem is the *left* field can only take a node value, but the only way to construct a node value is to apply the node constructor. Therefore, only an infinitely long sequence of node applications could construct such a value. FORMULA returns an error message like this:

```
(3, 4): The type Node is badly defined; it does not accept
        any finite terms.
```

The following domain defines nodes in two equivalent ways. It is accepted by the compiler:

```
domain Trees
{
   Node ::= (key  : Integer,
             left : Node + {NIL},
             right: Node + {NIL}).

   Node ::= (key: Integer, left: Tree, right: Tree).
   Tree ::= Node + {NIL}.
}
```

The type *Tree* is a super-type of the type *Node*, because in contains all node values and the additional value *NIL*.

2 Domains and Models

The purpose of a domain is describe a "class of things". The purpose of a model is to describe a specific "thing". Here are a few examples that we explore in this tutorial:

1. **DAG**: The DAG domain describes the properties of directs acyclic graphs (DAGs). A DAG model represents an individual DAG.
2. **SAT**: The SAT domain describes the set of satisfiable boolean expressions. A SAT model describes a single expression and the variable assignments that witness its satisfiability.
3. **FUNC**: The FUNC domain describes a small language of arithmetic functions and the rules of their evaluation. A FUNC model represents a program of the FUNC language.

The first step to define a "class of things" is to create a representation for "things" using FORMULA data types. Consider the classical definition of a directed graph G:

$$G \overset{def}{=} (V, E) \qquad \text{where } E \subseteq V \times V. \tag{1}$$

Classically, a directed graph is represented by a set of vertices V and set of edges E; each $e \in E$ is a pair of vertices. Furthermore, suppose vertices are represented by integers, then the set of all finite integer-labeled graphs is:

$$\mathcal{G} \overset{def}{=} \{(V, E) \mid V \subset \mathbb{Z} \land E \subseteq V \times V\} \qquad \text{where every } V \text{ is finite.}$$

Here is an example of a specific graph:

$$G_{ex} \overset{def}{=} (\{1, 2, 100\}, \{(1, 2), (100, 100)\}).$$

FORMULA does not directly support sets and relations, so we cannot express vertices and edges as sets. Instead, integer-labeled graphs are represented using two data constructors V and E as follows:

Example 1 (Integer-labeled graphs).

```
domain IntGraphs
{
   V ::= new (lbl: Integer).
   E ::= new (src: V, dst: V).
}
```

Intuitively, vertices are values such as:

```
V(1), V(2), V(100)
```

and edges are values such as:

```
E(V(1), V(2)), E(V(100), V(100))
```

Though these constructors provide representations for the elements of a graph, the domain does not define any specific graph elements. This is because the domain is intended as a schema for all graphs; it is not intended to represent a specific graph. A specific graph is represented by a model, as follows:

Example 2 (A small graph).

```
1: model Gex of IntGraphs
2: {
3:    V(1).
4:    V(2).
5:    V(100).
6:    E(V(1), V(2)).
7:    E(V(100), V(100)).
8: }
```

A model has a name and indicates the domain to which it belongs (line 1). The body of a model is a list of values separated by periods. These periods are actually assertions about the model. Each expression $f(\ldots)$. is an assertion:

"The value $f(\ldots)$ is always provable in the model M."

Thus, the model *Gex* contains a set of assertions, built with data constructors, about some vertex and edge values. These assertions define the specific elements of the graph *Gex*.

2.1 Querying Models

To understand how a model contains a set of assertions, create a file called `ex1.4ml` and copy the code contained in Examples 1 and 2. A *query operation* tests if a property is provable on a model. Follow these steps to test if a vertex called $V(1)$ exists in the model *Gex*:

```
1:  ...\Somewhere>Formula.exe
2:
3:  []> load ex1.4ml
4:  (Compiled) ex1.4ml
5:  0.82s.
6:  []> query Gex V(1)
7:  Started query task with Id 0.
8:  0.06s.
9:  []> ls tasks
10:
11: All tasks
12:  Id | Kind  | Status | Result |      Started      | Duration
13: ----|-------|--------|--------|-------------------|----------
14:  0  | Query |  Done  |  true  | 5/17/2013 3:28 PM |  0.04s
15: 0.03s.
```

Line 3 loads and compiles the file `ex1.4ml`. Line 6 starts a query operation on the model *Gex* and tests for the property $V(1)$. If $V(1)$ is provable in *Gex* then this query operation returns true; otherwise it returns false. Starting a query spawns a new background task that may take some time to complete. The Id of the newly created task is reported (e.g. Id 0). Line 9 causes the status of all tasks to be displayed. Line 14 shows that the query completed with the result true.

On the other hand this query evaluates to false, because there is no vertex named $V(3)$:

```
[]> query Gex V(3)
```

This is very important: Any query that is not provable using the model (and its domain declarations) is false. The query $V(3)$ is false for the model *Gex* because there are no assertions that can prove it. We can ask more interesting queries, such as: Does there exist some vertex?

```
[]> query Gex V(x)
```

This query contains a *variable* called x. More generally, a query is true if there is some substitution for the variables that is provable. This query is true because replacing x with 1, 2, or 100 forms queries that are provable. Variables appearing in a query are not declared and are local to the query expression.

More complicated patterns can be used in a query. This one tests if there is an edge that loops back onto the same vertex.

```
[]> query Gex E(x, x)
```

It is true because if $x = V(100)$ then $E(V(100), V(100))$ is provable. As with any language, it is possible to mistype commands. The FORMULA type system will catch some of these mistakes. For example, the query

```
[]> query Gex E(V(x), x)
```

is always false because x must simultaneously be an integer and a vertex, which is never possible. In this case the query is ignored and warning messages are returned:

```
[]> qr Gex E(V(x), x)
commandline.4ml (2, 1): Argument 2 of function E is badly typed.
commandline.4ml (0, 0): The install operation failed
Failed to start query task.
0.02s.
[]>
```

Queries can be conjoined using the comma operator (‘,’). Such a query is true if there is some substitution of variables making every conjunct true. This query tests if there are two edges that can be placed end-to-end:

```
[]> query Gex E(x, y), E(y, z)
```

Notice that the variables x and z only appear once in the query. Variables that only appear once can be written with an underscore (‘_’); every occurrence of an underscore creates a new variable with a different name from all other variables in the expression. The previous query can be rewritten as:

```
[]> query Gex E(_, y), E(y, _)
```

Underscores are useful for visually emphasizing those variables that appear in multiple places. Queries can also be formed from built-in relations and constraints.

```
[]> query Gex V(x), x > 20
```

Constructor labels can be used to write more readable queries. The query:

“Is there an edge whose source vertex has a label greater than 100?”

can be written as:

```
[]> query Gex e is E, e.src.lbl > 100
```

The constraint e *is* E requires e to be a provable value of type E.

The order in which conjuncts are written does not matter; the semantics of a query is the same. However, queries can only check for properties that can be answered using model assertions and a finite number of evaluations. For example, this query is not allowed:

```
[]> query Gex x > 100
```

It asks if there exists a number greater than 100. While the answer is “yes”, it cannot be proved using the assertions written in the model, nor can it be proved by evaluating > for a finite number of values. In this case, the following message is returned:

```
[]> query Gex x > 100
commandline.4ml (2, 3): Variable x cannot be oriented.
commandline.4ml (0, 0): The install operation failed
Failed to start query task.
0.02s.
[]>
```

The message `''Variable x cannot be oriented''` means the compiler cannot express x in such a way that the query can be evaluated. Note that FORMULA can reason about the properties of numbers, but the query operation is not the mechanism to accomplish this. We shall discuss this more in Section ??.

Finally, it is important to remember that even though $f(\ldots,t,\ldots)$ might be provable, this does not imply that t is itself provable. Consider the following model:

```
model NoVertices of IntGraphs
{
   E(V(1), V(2)).
}
```

This query is true:

```
[]> query NoVertices E(_,_)
```

But this query is false:

```
[]> query NoVertices V(_)
```

There are no provable vertices, even though a vertex value does appear within a provable edge. (This may surprise users familiar with term-rewriting systems.)

2.2 Model Conformance

A model M *conforms* to a domain D if the following properties are satisfied:

P1. Every assertion in M is constructed from *new-kind* constructors.
P2. Every constructor application in M obeys the type declarations in D.
P3. The operation `query M D.conforms` evaluates to true.

P1-P2 are checked whenever a model is compiled; any violation causes a compile-time error; P3 is checked upon request. A *new-kind* constructor is a constructor marked with the modifier *new*. Recall that both V and E constructors were marked with this modifier.

```
V ::= new (lbl: Integer).
E ::= new (src: V, dst: V).
```

Constructors that are not marked with the *new* modifier are only used to perform auxiliary computations; they can never appear in models. (We demonstrate this in more detail in later sections.) P2 is familiar from earlier examples. It is always illegal to use constructors with badly typed arguments, as in:

```
model BadlyTyped of IntGraphs
{
   E("Foo", "Bar").
}
```

P3 allows domains to place fine-grained constraints on the conformance relationship. Unlike the first two properties, satisfying P3 can be difficult and evaluating its satisfaction can be expensive. For these reasons, FORMULA only checks P3 upon request and failure of this property is not an error.

2.3 Relational Constraints

We began by trying to express the set of all integer-labeled finite graphs. The intent was to define *IntGraphs* so its set of conforming models would be equivalent to the set of integer-labeled finite graphs. Properties P1-P2 guarantee that models contain only vertex and edge assertions, which encode graph elements in an obvious way. However, there is the additional constraint that graph edges should be pairs of graph vertices, i.e. $E \subseteq V \times V$. Did we capture this constraint correctly? The answer is that it depends on how we choose to define the set of vertices V present in a model M. There are two options: (1) Every occurrence of a vertex value anywhere in the model is implicitly a member of V. (2) Only those vertices that are provable are members of V. Consider this example:

```
model SomeVertices of IntGraphs
{
   V(1).
   E(V(1), V(2)).
}
```

Under the first definition, the vertex set for this model is $\{1,2\}$ and the edge set is $\{(1,2)\}$. This satisfies the constraint $E \subseteq V \times V$. Under the second definition the vertex set is only $\{1\}$ and the edge set violates the constraint because $2 \notin V$. By default, FORMULA uses the second and more restrictive definition: $E \subseteq V \times V$ means every argument to $E(,)$ must be provable. Models violating this property do not conform to the domain. These kinds of constraints are so common that FORMULA automatically introduces them. To see this, add the code for *SomeVertices* to `ex1.4ml` and run this query:

```
[]> query SomeVertices IntGraphs.conforms
```

The result of this query is false because $V(2)$ is not derivable. Add $V(2)$ to the model, save it, and type:

```
[]> reload ex1.4ml
```

Evaluate the query again to observe that it evaluates to true.

We call the previous kind of constraint a *relational constraint*. Relational constraints are injected by the compiler when it appears that one constructor is being used to encode a set S and another constructor is being used to encode a relation $R \subseteq \ldots \times S \times \ldots$.

Definition 21 (Relational Constraint)**.** The constructor R is relational on constructor S in position i if the constructor R is declared as:

```
R ::= new (..., arg_i: T_i, ...).
```

and S is a subtype of T_i. The relational constraint means that for every provable value t containing $R(\ldots, t_i, \ldots)$ and $t_i = S(\ldots)$ then t_i must also be provable.

Some type declarations are not intended to encode finite relations, and the default behavior would produce strange results. Consider the following recursive definition for binary trees:

```
Node ::= new (left: Node + {NIL}, right: Node + {NIL}).
```

This declaration fits the pattern for relational constraints and the generated constraints are satisfiable. However, it assumes the user intended to encode a relation with the following strange property:

$$Node \subseteq Node \times Node.$$

The only finite relation satisfying this property is the empty set. There is clearly a semantic mismatch between the binary relation $Node$ and the binary data constructor $Node$. In order to bring attention to this mismatch FORMULA produces the following error:

```
ex.4ml (3, 4): The constructor Node cannot have relational
               constraints on itself; see argument 1.
ex.4ml (3, 4): The constructor Node cannot have relational
               constraints on itself; see argument 2.
```

When this error occurs at position i, the user must explicitly indicate that the constructor is not intended to encode a relation at position i. Placing the *any* modifier before the argument type indicates that *any* well-typed value is permitted here, not just those values that are provable. The compiler does not generate relational constraints for this argument and the error message does not occur. Here is the proper declaration for the recursive binary tree constructor:

```
Node ::= new (left: any Node + {NIL}, right: any Node + {NIL}).
```

Of course, binary trees can also be encoded using finite relations, but the encoding is a different one from the recursive constructor shown above.

2.4 Finite Functions

Finite functions are a special case of finite relations satisfying additional constraints. Consider the representation of a forest F of binary trees as follows:

$$F \overset{def}{=} (V, parent) \qquad \text{where } parent : V \to \{\top\} \cup (\{L, R\} \times V).$$

The set V contains the vertices of the forest and the function *parent* assigns to each vertex v its parent: $parent(v) = (L, u)$ if v is the left child of u; $parent(v) = (R, u)$ if v is the right child of u; $parent(v) = \top$ if v is a root. Additionally, the *parent* function should not introduce a cycle, but we ignore this constraint for now. The parent function can be treated as a finite relation by listing its input-output pairs. Consider this forest where 1 is a root, 2 is its left child, and 3 is its right child:

$$F_{ex} \overset{def}{=} (\{1, 2, 3\}, \{(1, \top), (2, (L, 1)), (3, (R, 1))\}).$$

The *parent* relation encodes a total finite function if it is total on the domain:

$$\forall v \in V.\ \exists x.\ (v, x) \in parent,$$

and every input is related to a unique output:

$$\forall v \in V.\ \forall x, y.\ (v, x) \in parent \wedge (v, y) \in parent \Rightarrow x = y.$$

FORMULA supports declarations of finite functions also. As before, these declarations are really introducing data constructors plus additional constraints that provable values must encode finite functions. Example 3 shows the syntax.

Example 3 (Relational Trees).

```
domain RelTrees
{
   V       ::= new (lbl: Integer).
   Parent  ::= fun (chld: V => cxt: any {ROOT} + Context).
   Context ::= new (childPos: {LFT, RT}, prnt: V).
}

model Fex of RelTrees
{
   V(1). V(2). V(3).
   Parent(V(1), ROOT).
   Parent(V(2), Context(LFT, V(1))).
   Parent(V(3), Context(RT, V(1))).
}
```

Line 5 declares the *Context* constructor to represent $\{L, R\} \times V$. Line 4 declares the *Parent* constructor using the *fun* modifier. This modifier implies *new*. The arguments on the left side of => correspond to the domain of the relation and the arguments on the right side correspond to the codomain. The codomain contains the value *ROOT* (i.e. $\top$) and any *Context* value. The *fun* modifier injects uniqueness constraints, i.e. for all provable values $Parent(v, x)$ and $Parent(v, y)$ then $x = y$. The *totality arrow* => injects totality constraints, i.e. for every provable $V(x)$ there is a provable $Parent(V(x), y)$. Totality is affected by the *any* modifier. If an argument is marked with *any*, then the finite function must be defined for every well-typed value. For example, this declaration would cause an error:

```
Parent  ::= fun (chld: any V => cxt: any {ROOT} + Context).
```

The *any* modifier applied to the first argument means there must be a provable $Parent(V(x), y)$ value for every well-typed value $V(x)$. This implies an infinite

Table 4. Table of relation / function modifiers

Relation / Function Modifiers	
Syntax	Meaning
`R ::= new (..., T_i, ...).`	Relational constraint: If S is a constructor, $S <: T_1$, $R(\ldots, t_i, \ldots)$ occurs in a provable value, and $t_i = S(\ldots)$, then t_i must be provable.
`R ::= new (..., any T_i, ...).`	Occurrences of R are exempt from the relational constraint in position i.
`R ::= fun (..., D_m -> ..., C_n).`	Partial function: Same as `new`. Additionally, the set of provable R values must form a partial function from $D_1 \times \ldots \times D_m$ to $C_1 \times \ldots \times C_n$.
`R ::= fun (..., D_m => ..., C_n).`	Total function: Same as partial function, but must be total on $D_1 \times \ldots \times D_m$; totality is modified by `any`.
`R ::= inj (..., D_m -> ..., C_n).`	Partial injection: Same as partial function; additionally if $R(\boldsymbol{x}, \boldsymbol{z})$ and $R(\boldsymbol{y}, \boldsymbol{z})$ are provable then $\boldsymbol{x} = \boldsymbol{y}$.
`R ::= inj (..., D_m => ..., C_n).`	Total injection: Constrained to be a total function and partial injection.
`R ::= sur (..., D_m -> ..., C_n).`	Partial surjection: Same as partial function; additionally must be total on $C_1 \times \ldots \times C_n$ (totality is modified by `any`).
`R ::= bij (..., D_m -> ..., C_n).`	Bijection: Constrained to be a total surjection and partial injection; partiality arrow has no effect.
`R ::= bij (..., D_m => ..., C_n).`	Bijection: Constrained to be a total surjection and partial injection.

number of provable values, which is not permitted. FORMULA returns the following error message:

```
ex.4ml (4, 4): The function Parent requires totality on an
               argument supported by an infinite number of
               values; see argument 1.
```

Whenever the domain of a finite function is infinite that function cannot be total, but it can be *partial*. The *partial arrow* −> indicates a partial function, which need not be defined on every element of its domain. Suppose every vertex label should also be given a "pretty name". This could be expressed by extending the signature of V as follows:

```
V ::= fun (lbl: Integer -> prettyName: String).
```

Every vertex must have a unique pretty name, but there does not need to be a vertex defined for every integer. Table 4 shows the complete variety of relation / function modifiers.

2.5 Recursive Types, Aliases, and Symbolic Constants

Example 3 showed a partial specification of binary trees using a representation inspired by finite relations. Another approach is to use the full power of recursive data types. In this representation an entire tree is a single complex value. The locations of values distinguishes vertices from each other. This is in contrast to using of unique identifiers to distinguish vertices.

Example 4 (Algebraic Trees).

```
1:  domain AlgTrees
2:  {
3:     Node ::= new (left:  any Node + {NIL},
4:                   right: any Node + {NIL}).
5:     Root ::= new (root:  any Node).
6:  }
7:
8:  model Fex' of AlgTrees
9:  {
10:    Root(
11:       Node(
12:            Node(NIL, NIL),
13:            Node(NIL, NIL))).
14: }
```

Example 4 shows an algebraic representation of trees using data constructors. Notice that the entire tree is a single value (line 10). The *Root*() constructor is used to mark some nodes as roots in the forest. The left and right children of the root are both nodes without children, represented by the value $Node(NIL, NIL)$ (lines 12, 13). In fact, both these nodes are exactly the same value. They are distinguishable by where they occur in the construction of the parent. This representation has several advantages: (1) Nodes do not need to be labeled. (2) It is impossible to create an illegal tree. (3) Two trees are the same if and only if they are the same value. Also, because the same value can represent many nodes, it is possible to define the value once and reuse it in many places. Reuse is accomplished by introducing an *alias* as follows:

```
leaf is Node(NIL, NIL).
```

The right-hand side of the *is* keyword requires a constructed assertion. The left-hand side is an identifier that stands for the constructed value. Using aliases, the previous model can be expressed as:

```
model Fex_Shared of AlgTrees
{
   leaf is Node(NIL, NIL).
   Root(Node(leaf, leaf)).
}
```

Aliases can be used to represent models with exponentially less space. Consider the following complete binary tree with 1,023 nodes:

```
model BiggerTree of AlgTrees
{
   leaf is Node(NIL, NIL).

   subtree_3    is Node(leaf       , leaf).
   subtree_7    is Node(subtree_3  , subtree_3).
   subtree_15   is Node(subtree_7  , subtree_7).
   subtree_31   is Node(subtree_15 , subtree_15).
   subtree_63   is Node(subtree_31 , subtree_31).
   subtree_127  is Node(subtree_63 , subtree_63).
   subtree_255  is Node(subtree_127, subtree_127).
   subtree_511  is Node(subtree_255, subtree_255).
   subtree_1023 is Node(subtree_511, subtree_511).

   Root(subtree_1023).
}
```

Aliases are visible to all assertions within their defining model. The order in which aliases are defined is inconsequential. However, aliases definitions cannot form as cycle as this would be equivalent to applying an infinite number of constructors. This property is checked at compile time. Here is an example:

```
model InfiniteTree of AlgTrees
{
   infinite_left  is Node(infinite_right, NIL).
   infinite_right is Node(NIL, infinite_right).
   Root(Node(infinite_left, infinite_right)).
}
```

The *infinite_left* node has an infinitely deep subtree as its left node, and the *infinite_right* node has an infinitely deep subtree as its right node. This problem is detected and the following error is reported.

```
inf.4ml (4, 4): Symbolic constant InfiniteTree.%infinite_right is
                defined using itself.
inf.4ml (3, 4): Symbolic constant InfiniteTree.%infinite_left is
                defined using itself.
```

2.6 Symbolic Constants

The previous error messages referred to *symbolic constants*. A symbolic constant is a *constant*, i.e. it is a function that takes no arguments and returns a value. However, the value it returns may not be known at compile time. This latter property is unlike the constants we have encountered so far. For example, the constant 1 always returns the constant 1 and the constant *NIL* always returns the constant *NIL*. Each model alias a declared in model M defines a symbolic constant called $M.\%a$. It returns the value produced by expanding all aliases in the model. Consider Example 2 rewritten with aliases:

```
model Gex' of IntGraphs
{
   v1   is V(1).
   v2   is V(2).
   v100 is V(100).

   e_1_2     is E(v1, v2).
   e_100_100 is E(v100, v100).
}
```

These aliases create symbolic constants with the following properties:

$$\begin{aligned}
&Gex'.\%v1 &&= V(1)\\
&Gex'.\%v2 &&= V(2)\\
&Gex'.\%v100 &&= V(100)\\
&Gex'.\%e\_1\_2 &&= E(V(1), V(2))\\
&Gex'.\%e\_100\_100 &&= E(V(100), V(100))
\end{aligned}$$

Unlike variables, symbolic constants can be used to test the properties of specific model elements. Whenever a symbolic constant occurs in a query, it matches the corresponding assertion in the model where it was defined. For example:

```
[]> query Gex' Gex'.%v1.lbl = 1
```

This query is satisfied if the value represented by $Gex'.\%v1$ is provable and its label is equal to 1. This query evaluates to true. Symbolic constants are prefixed with the percent-sign ('%') to distinguish them from variables. Consider this query:

```
[]> query Gex' E(%v1, v1)
```

Here $\%v1$ is the same symbolic constant as before, but $v1$ is a variable. (Names do not need to be fully qualified if they can be unambiguously resolved.) This query is satisfied if there is some value for $v1$ that makes $E(V(1), v1)$ provable. The assignment $v1 = V(2)$ is one such value and the query evaluates to true. Finally, this query evaluates to false, because there is not an edge from $V(1)$ to $V(1)$:

```
[]> query Gex' E(%v1, %v1)
```

2.7 Separate Compilation

A FORMULA module can refer to modules in other files. The compiler will load and compile all files required to completely compile a program. There are several ways to refer to modules in another files. The first way is to qualify the module name with a source file using the *at* operator. For example:

```
model M of D at "foo.4ml" { ... }
```

The compiler will look for a domain named D in the file *foo.4ml*. This method does not affect the resolution of any other occurrence of D. For example, this code loads two different domains, both called D, from different files:

```
model M1 of D at "..\\..\\version1.4ml" { ... }
model M2 of D at '"..\..\version2.4ml"' { ... }
```

(Relative paths are resolved w.r.t. to the path of file where they occur.) However, it is illegal to define two modules with the same name in the same file.

```
model M of D at "version1.4ml" { ... }
model M of D at "version2.4ml" { ... }
```

This causes an error:

```
ex.4ml (2, 1): The module M has multiple definitions.
               See ex.4ml (1, 1) and ex.4ml (2, 1)
```

The disadvantage of the *at* operator is that it must be used on every reference to D. Another option is to register D using a *configuration block*. Configuration blocks have access to various configuration objects, one of which is called *modules* mapping names to files. To register D and D' in the scope of the file write:

```
1: [
2:    modules.D  = "D  at foo.4ml",
3:    modules.D' = "D' at foo.4ml"
4: ]
5:
6: model M of D { ... }
```

Lines 1 - 4 form a configuration block. Now every occurrence of D will resolve to D located in *foo.4ml*. If D is defined to be in several locations, then an error occurs.

Causes an error because D is defined to be at two different places.

```
[
   modules.D = "D at version1.4ml",
   modules.D = "D at version2.4ml"
]
```

Finally, each module has its own local configuration parameters. Modules inherit file level configurations, and can extend these with additional parameters. Module-level configurations are isolated from each other, so occasionally it may be useful to register domains at this level. Here is an example:

```
model M1 of D
[
   modules.D = "D at version1.4ml"
]
{
   ...
}

model M2 of D
[
   modules.D = "D at version2.4ml"
]
{
   ...
}
```

The module-level configuration block must be placed after the module declaration and before the opening curly brace. These configurations do not conflict, because they are lexically scoped to the modules $M1$ and $M2$. Separate compilation is available for all types of FORMULA modules.

3 Rules and Domain Constraints

Domain constraints are essential for defining “classes of things”. So far we have demonstrated type constraints and a few kinds of relation / function constraints (that also appear in type declarations). However, these constraints are not general enough to capture many properties. Recall the relational trees example

(Example 3) where we neglected to constrain trees to be acyclic. There was no way to write this constraint using the mechanisms presented thus far, and so the specification is incomplete. Some models that are not trees conform to the *RelTrees* domain.

The remaining aspects of the FORMULA language deal with computing properties of models using conditional statements, which we call *rules*. These computed properties can be used to write powerful domain constraints (among other things). A basic rule has the following syntax:

```
head :- body.
```

The *body* part can contain anything a query can contain. The *head* part is a sequence of constructor applications applied to constants and variables. A rule means:

"Whenever the body is provable for some variable assignment, then the head is also provable for that variable assignment."

Thus, a rule is a logical statement, but it can also be *executed* (like a query) to grow the set of provable values.

Consider the problem of computing the ancestors between vertices in a *RelTrees* model. First, we introduce a helper constructor called *anc*(,) into the *RelTrees* domain to represent the ancestor relationship:

```
anc ::= (ancestor: V, descendant: V).
```

Notice that *anc* is not modified with *new* (or any other modifier that implies *new*). This means an *anc* value can never be asserted by a model. The only legal way to prove an *anc* value is by proving its existence with rules. We call *anc* a *derived-kind* constructor. Here is a rule that can prove u is an ancestor of w if u is the parent of w:

```
anc(u, w) :- Parent(w, Context(_, u)).
```

Here is a rule that can prove u is an ancestor of w if u is an ancestor v and v is an ancestor of w.

```
anc(u, w) :- anc(u, v), anc(v, w).
```

As with queries, the comma operator (',') conjuncts constraints. Every conjunct must be satisfied for the rule to be satisfied. If two rules have the same head, then they can be syntactically combined using the semicolon operator (';').

```
anc(u, w) :- Parent(w, Context(_, u)); anc(u, v), anc(v, w).
```

Intuitively the semicolon operator behaves like disjunction, but remember that each semicolon actually marks the body of an independent rule.

Table 5. Table of matching constraints

Matching Constraints	
Syntax	Meaning
`q`	true if the derived constant q is provable; false otherwise.
`f(t1,...,tn)`	true for every assignment of variables where $f(t_1,\ldots,t_n)$ is provable; false otherwise.
`x is f(t1,...,tn)`	true for every assigment of variables where $f(t_1,\ldots,t_n)$ is provable and $x = f(t_1,\ldots,t_n)$; false otherwise.
`x is T`	true if x is a provable value and a member of the type named T; false otherwise.

Table 6. Table of interpreted predicates

Interpreted Predicates	
Syntax	Meaning
`no {...}`	true if the set comprehension is empty; false otherwise.
`t = t'`	true if t and t' are the same value; false otherwise.
`t != t'`	true if t and t' are different values; false otherwise.
`t < t'`	true if t is less than t' in the order of values; false otherwise.
`t <= t'`	true if t is less than equal to t' in the order of values; false otherwise.
`t > t'`	true if t is greater than t' in the order of values; false otherwise.
`t >= t'`	true if t is greater than or equal to t' in the order of values; false otherwise.
`x : T`	true if x is a member of the type named T; false otherwise.

3.1 Derived Constants

A tree-like graph is not a tree if it has a cycle in its ancestor relation. There is a cycle if and only if some vertex is an ancestor of itself. The following rule summarizes this property:

```
hasCycle :- anc(u, u).
```

The symbol *hasCycle* is a *derived-kind* constant (or just *derived constant*). A derived constant is declared by using it, unqualified, on the left-hand side of some rule. Rules can only prove derived constants or values built from constructors. This rule causes an error:

```
1 :- V(_).
```

```
ex.4ml (1, 4): Syntax error - A rule cannot produce
              the base constant 1
```

The full name of a derived constant c declared in domain D is $D.c$. This allows for (an optional) coding idiom where each derived constant is declared in exactly one place.

```
Declares hasCycle, because appears unqualified on left-hand side of the rule.
(Does not affect the provable values.)
hasCycle :- RelTrees.hasCycle.
First use of hasCycle; not a declaration because used in qualified form.
RelTrees.hasCycle :- anc(u, u).
Second use of hasCycle; this rule is redundant.
RelTrees.hasCycle :- anc(u, w), anc(w, u).
```

3.2 Rule Bodies

The body of a rule is a conjunction of constraints. Constraints can be either *matching constraints* or *interpreted predicates*. A matching constraint is satisfied if there is some substitution of the variables where the resulting value is provable. An interpreted predicate is satisfied if there is some substitution of the variables where a special predicate evaluates to true. Tables 5 and 6 list the forms of matching constraints and interpreted predicates. The order in which conjuncts appear is irrelevant; as with queries, all variables must be orientable.

To demonstrate interpreted predicates, consider that our *RelTrees* domain also allows a single node to have more than two left (or right) children.

```
model TwoLeftChildren of RelTrees
{
   v1 is V(1). v2 is V(2). v3 is V(3).
   Parent(v1, ROOT).
   Parent(v2, Context(LFT, v1)).
   Parent(v3, Context(LFT, v1)).
}
```

The following rule can be used to detect these anomalous graphs:

```
tooManyChildren :-
   Parent(x, Context(pos, parent)),
   Parent(y, Context(pos, parent)),
   x != y.
```

The disequality predicate (‘!=’) is used to detect if more than one distinct node has been assigned to the same position in the parent. Using the equality predicate (‘=’) and the selector operator (‘.’), the above rule can be rewritten as:

```
tooManyChildren :-
   px is Parent,
   py is Parent,
   px.chld != py.chld,
   px.cxt = py.cxt,
   px.cxt != ROOT.
```

The choice of whether to use the first or second rule is a matter of style. (Other equivalent rules can be written too.)

3.3 Interpreted Functions

FORMULA also provides many *interpreted functions*, such as $+$, $and(,)$, $toString()$, etc... An interpreted function can appear anywhere a t, t' or t_i appears in Tables 5 and 6. (See Section 5 for the full list of interpreted functions.) This constraint finds all the vertices whose child labels sum to the label of the parent.

```
SumToParent(v) :-
    Parent(x, Context(_, v)),
    Parent(y, Context(_, v)),
    x.lbl + y.lbl = v.lbl.
```

The interpreted function $+$ represents ordinary arithmetic addition, which is only defined for numeric values. Therefore $x.lbl$ and $y.lbl$ must be numeric values for the operation to be meaningful. These extra requirements are automatically added as *side-constraints* to the rule body. The complete rule constructed by the FORMULA compiler is:

```
SumToParent(v) :-
    Parent(x, Context(_, v)),
    Parent(y, Context(_, v)),
    x.lbl + y.lbl = v.lbl,
    xlbl = x.lbl, xlbl : Real,
    ylbl = y.lbl, ylbl : Real.
```

These side-constraints have an important consequence: A rule will never evaluate operations on badly-typed values. The side-constraints guarantee that the rule is not triggered for values where the operators are undefined.

To demonstrate this, create a new file containing the code in Example 5:

Example 5 (Pretty labeled trees).

```
domain PrettyRelTrees
{
    V       ::= new (lbl: Integer + String).
    Parent  ::= fun (chld: V => cxt: any {ROOT} + Context).
    Context ::= new (childPos: {LFT, RT}, prnt: V).

    SumToParent ::= (V).
    SumToParent(v) :-
        Parent(x, Context(_, v)),
      Parent(y, Context(_, v)),
      x.lbl + y.lbl = v.lbl.
}

model StringTree of PrettyRelTrees
{
   vA is V("a").
   vB is V("b").
   vC is V("c").

   Parent(vB, ROOT).
   Parent(vA, Context(LFT, vB)).
   Parent(vC, Context(RT, vB)).
}
```

In this domain vertices can have either integer labels or “prettier” string labels. The model *StringTree* contains a single tree with only string labels. Run the query:

```
[]> query StringTree SumToParent(_)
```

and notice that it evaluates to false. The query does not produce any errors or exceptions. The side-constraints created by + simply prevents the rule from being triggered by the string labeled vertices in *StringTree*.

3.4 Type Environments

In fact, FORMULA uses type inference to inform you about the values that may trigger a rule. Type:

```
[]> types PrettyRelTrees
```

and the following listing is returned:

```
+ Type environment at (8, 4)
  v: V(Integer)
  ~dc0: Parent(V(Integer), Context({RT} + {LFT}, V(Integer)))
  x: V(Integer)
  ~dc1: {RT} + {LFT}
  ~dc2: Parent(V(Integer), Context({RT} + {LFT}, V(Integer)))
  y: V(Integer)
  ~dc3: {RT} + {LFT}
  ~sv0: Integer
  ~sv1: Integer
  ~sv2: Integer
   + Type environment at (8, 22)
     v: V(Integer)
     ~dc0: Parent(V(Integer), Context({RT} + {LFT}, V(Integer)))
```

```
15:     x: V(Integer)
16:     ~dc1: {RT} + {LFT}
17:     ~dc2: Parent(V(Integer), Context({RT} + {LFT}, V(Integer)))
18:     y: V(Integer)
19:     ~dc3: {RT} + {LFT}
20:     ~sv0: Integer
21:     ~sv1: Integer
22:     ~sv2: Integer
```

This listing shows the inferred types for variables. Type inference is displayed as a sequence of nested *type environments*. The first type environment (line 1) lists the inferred types for variables in the head of the rule. The second nested type environment (line 12) lists the inferred types for the body of the rule. Had the rule contained several bodies via the semicolon operator, then there would be a distinct type environment for each body, and all of these environments would be nested under the head. The type of the head would then be the union of all the types in the bodies.

Line 2 shows that the variable v is guaranteed to be of type $V(Integer)$; it can never be a string labeled vertex. The variables named *~dc0*, *~dc1*, *~dc2*, *~dc3* are *don't care* variables. They were generated by the compiler wherever we preferred not to provide a variable name. This happened in four places in the body:

```
Parent(x, Context(_, v)), Parent(y, Context(_, v))
```

In the first matching constraint we did not provide a variable to bind the matching constraint (using the *is* operator). The compiler generates one called *~dc0* and its type is:

```
Parent(V(Integer), Context({RT} + {LFT}, V(Integer)))
```

This matching constraint is guaranteed to match only *Parent* values whose child is an integer-labeled vertex in the context of an integer-labeled parent. The variable *~dc1* occurs because we used the underscore operator ('_') in the first argument of *Context*. The variables *~dc2* and *~dc3* were analogously created for the second pattern. Finally, the variables *~sv0*, *~sv1*, and *~sv2* are compiler-generated *selector variables*. They stand for the selection of fields:

```
~sv0 = x.lbl, ~sv1 = y.lbl, ~sv2 = z.lbl
```

Examining the type environments is useful for understanding how constraints in the body interact to restrict the triggering of rules. FORMULA only generates an error if it is impossible to trigger a rule. For instance, change the declaration of V to:

```
V ::= new (lbl: String).
```

and reload the program. This will result in errors:

```
ex.4ml (8, 80): Argument 1 of function + is badly typed.
ex.4ml (8, 80): Argument 2 of function + is badly typed.
```

3.5 Set Comprehensions

Sometimes it is necessary to aggregate *all* the provable values of a given type into a single result. The rules we have shown so far cannot accomplish this task. Consider again the *IntGraphs* example (Example 1) and imagine trying to compute the *in-degree* of a vertex, i.e. the number of distinct edges coming into a vertex. One might be tempted to write rules such as:

```
indeg_atleast_2(v) :-
   v is V,
   e1 is E(_, v), e2 is E(_, v),
   e1 != e2.

indeg_atleast_3(v) :-
   v is V,
   e1 is E(_, v), e2 is E(_, v), e3 is E(_, v),
   e1 != e2, e1 != e3, e2 != e3.
```

These rules can only determine lower bounds on the in-degree by testing for vertices with at least k distinct incoming edges. Using this approach, we must write a rule for every possible in-degree that could be encountered. (We could never write all such rules.) Also, the rules would get larger; the rule computing *indeg_atleast_k* would contain $O(k^2)$ constraints. Also, even with these two rules, we still cannot write a rule that finds vertices whose in-degree is exactly two. A vertex v has a degree of exactly two if *indeg_atleast_2*(v) is provable and *indeg_atleast_3*(v) is *not* provable. But so far we cannot test if *indeg_atleast_3*(v) is *not* provable in the body of a rule.

Set comprehensions remedy this problem and fundamentally increase the expressive power of FORMULA. A set comprehension has the following form:

```
{ t1, ..., tn | body }
```

The body part can be legal rule body (e.g. it can contain the semicolon operator and nested comprehensions). The expressions $t_1, \ldots, t_n$ can be any legal combination of constants, constructors, variables, and selectors. A set comprehension means:

"Form a set S of values as follows: For every assignment of variables satisfying the body substitute these values in to each t_i and add each t_i to the set S.

Here is an example of using a set comprehension to compute the in-degree of an arbitrary vertex.

```
indeg(v, k) :- v is V, k = count({ e | e is E(_, v)}).
```

The *count*() operation is an interpreted function that takes a set comprehension and returns its size. The comprehension forms a set of all the edges with v in the destination position, and k is equal to the size of this set. Notice that the comprehension sees the variables declared outside of it, and so each choice of value for the variable v instantiates v inside the comprehension. However, variables introduced inside the comprehension do not escape. The variable e is scoped to the comprehension and other comprehensions cannot see it. Consider this rule:

```
q :- count({x | V(x), x > 0}) = count({x | V(x), x < 0}).
```

The first occurrence of x is lexically scoped to the first comprehension, and second occurrence is scoped to the second comprehension. This rule tests if the number of vertices with positive labels equals the number with negative labels. This rules *does not* constrain x to be both positive and negative (which is impossible).

Comprehensions can only be used in combination with certain interpreted predicates / interpreted functions. Variables can only be assigned values, so it is illegal to assign a set comprehension to a variable. The interpreted predicate *no* is used to test if a comprehension is empty, it is equivalent to the constraint:

```
count({ ... }) = 0
```

This rule computes all the *sources* in a graph, i.e. all the vertices with zero in-degree.

```
source(v) :- v is V, no { e | e is E(_, v) }.
```

The body of this comprehension consists of a single matching constraint. For the special case where *no* is applied to a comprehension with a single matching constraint, then only the body of the comprehension needs to be written.

```
source(v) :- v is V, no e is E(_, v).
```

Equivalently,

```
source(v) :- v is V, no E(_, v).
```

3.6 General Rules and Rule Heads

A general rule has for the form:

```
head1, ..., headm :- body1; ...; bodyn.
```

It is equivalent to the set of rules:

```
head1, ..., headm :- body1.
```

$\vdots$

```
head1, ..., headm :- bodyn.
```

A general rules proves all heads $head_1, \ldots, head_m$ for every satisfying assignment of the body. Each head must be formed from constants, constructors, and variables. A rule head must satisfy the following properties:

- Under all circumstances, a rule head must evaluate to a derived constant or a constructed value.
- Under all circumstances, a rule head must evaluate to a well-typed value.
- Every variable appearing in a rule head must appear at the top level of the body. (The variable cannot be introduced by a comprehension.)

All of these properties are checked at compile time and generate errors if violated. A rule violating the first property was demonstrated earlier (i.e. a rule that proved the constant 1). The second property is the most interesting. It requires the constraints in the body to prove that every possible assignment satisfying the body yields a well-typed head value. Imagine adding the following code to Example 5 where vertices can have integer or string labels:

```
1:    IntLabel ::= (Integer).
2:
3:    IntLabel(x) :- V(x).
4:    isIntLabel  :- IntLabel(x), V(x).
```

The rule in line 3 matches a vertex; the inferred type of x is $String + Integer$. However, the head $IntLabel(x)$ requires x to be an integer. If the rule were triggered by a string-labeled vertex, then it would produce a badly-typed head value. This danger is detected by the FORMULA compiler:

```
ex.4ml (3, 4): Argument 1 of function IntLabel is unsafe.
    Some values of type String are not allowed here.
```

Contrast this with the rule in line 4. Here the conjuncts $IntLabel(x)$ and $V(x)$ constrain x to be an integer. This rule is accepted by the compiler. Remember that the bodies must constrain variables enough so that the compiler can prove the heads are always well-typed. This architecture is designed to detect mistakes in rules at compile time.

There is one exception to the previous discussion. Selectors can be used in the head of a rule, in which case they are treated as if they occurred the body. This mean selectors appearing in the head will constrain the variables appearing in the body. This rule computes if one vertex is a child of another in a relational tree:

```
isChild(p.chld, p.cxt.prnt) :- p is Parent.
```

The occurrence of $p.cxt.prnt$ constrains p to have the type $Parent(V, Context)$ even though no such constraint appears directly in the body. The rule the compiler produces is actually:

```
isChild(p.chld, p.cxt.prnt) :-
   p is Parent, _ = p.chld, _ = p.cxt.prnt.
```

Finally, every variable appearing in a rule head must also appear at the top level of a rule body. This rule is illegal:

```
isChild(p.chld, x) :- p is Parent.
```

because x does not appear in the body. This restriction prevents rules from proving an infinite number of facts, thereby preserving executability of rules. More subtly, this rule is also illegal:

```
isChild(x, y) :- no Parent(x, Context(_, y)).
```

Intuitively, this rule succeeds if there is no *Parent* value matching the constraint, so no assignments are available for the variables x and y in the head. A more general way to understand the problem is to re-introduce the lexical scoping braces:

```
isChild(x, y) :- no { p | p is Parent(x, Context(_, y)) }.
```

The variables x and y are not visible outside of the set comprehension, so they cannot be used in the head. Finally, a *fact* is a rule whose head contains no variables and whose body is empty. It is treated as a rule whose heads are always provable. A fact can be written as:

```
value1, ..., valuem.
```

3.7 Stratification and Termination

Rules are a form of executable logic. This means we can simultaneously treat them as a set of logical statements or as a kind of program, but both points-of-view should agree on what the rules mean. Obtaining this agreement becomes complicated without additional requirements on the structure of rules. The first major challenge arises because of set comprehensions. Consider these rules:

```
p :- no q.
q :- no p.
```

To execute these rules as program, FORMULA (1) chooses a rule, (2) computes all the new values it proves, and (3) repeats steps 1-2 until no new values are proved. If the first rule is executed first then p is proved because q is not, but then q cannot be proved. On the other hand, if the second rule is executed first then q is proved because p is not, but then p cannot be proved. Consequently, the answer to `query M p` depends on the order of execution.

There are two ways to reconcile this behavior. Either we can treat this behavior as correct, in which case the logical interpretation of rules must be generalized. Or, we can restrict the structure of rules to prevent this behavior all together. We choose the second approach, and prevent rules that would exhibit the behavior just described. One well-known restriction that eliminates this problem is called *stratification.*

Definition 31 (Stratification)**.** A FORMULA program is stratified if there is no set comprehension that examines values (indirectly) proved by the rule containing the comprehension.

In the previous example the first rule contains a set comprehension *no* $\{q \mid q\}$. And, q values are proved by the second rule, which examines p values (under a set comprehension of its own). Therefore the first set comprehension examines values that could be indirectly proved by the rule containing it. This is a sign that the order of execution could yield different outcomes. FORMULA produces the following error message for the previous program:

```
ex.4ml (1, 9): A set comprehension depends on itself.
               Listing dependency cycle 0...
ex.4ml (1, 9): A set comprehension depends on itself.
               Dependency cycle 0
ex.4ml (1, 4): A set comprehension depends on itself.
               Dependency cycle 0
ex.4ml (2, 9): A set comprehension depends on itself.
               Dependency cycle 0
ex.4ml (2, 4): A set comprehension depends on itself.
               Dependency cycle 0
```

The error messages list the locations of the rules and the set comprehensions that form a dependency cycle. Stratification is fully checked at compile time and unstratified programs are rejected.

Rules are executed so that all values are proved before a dependent set comprehension is executed. This strategy always computes a unique result, which coincides with the logical interpretation of rules. Users may write rules in any syntactic order, but they will always be executed to respect the dependencies of set comprehensions. For example:

```
smallInDegree :- no { v | indeg(v, k), k > 3 }.
indeg(v, k)   :- v is V, k = count({ e | e is E(_, v)}).
```

A graph has a “small in-degree” if no in-degree is greater than three. The first rule computes *smallInDegree* by comprehending over all the *indeg* values. Furthermore, the second rule computes *indeg* values by comprehending over all the edges. The order of execution will be: compute all the E values, and then compute all the *indeg* values, and then compute the *smallInDeg* value. It does not matter that *smallInDeg* appeared earlier in program text.

The second challenge with executable logic is *termination*.

Definition 32 (Termination)**.** A domain is terminating if the set of provable values is finite for every model of that domain.

A non-terminating domain may execute forever when evaluating a query. Currently FORMULA does not check for termination, so a user may write a non-terminating program and later find that query execution never stops. Theoretically termination cannot be checked with certainty for arbitrary programs, though many conservative analyses are possible and FORMULA may use some of them in the future. Here is a classic example of a non-terminating rule representing the *successor function* $s()$.

```
s(x) :- x = 0; x is s.
```

The value $s(0)$ is provable, and so the value $s(s(0))$ is provable, and so the value $s^n(0)$ is provable for every positive integer n. In fact, the successor function is one way to represent the natural numbers. Users should not try to axiomatize theories, such as the theory of natural numbers, using FORMULA. Instead, they should utilize the interpreted functions that already embed these theories into the FORMULA language.

3.8 Complex Conformance Constraints

We have shown how rules can compute properties of models. To create a conformance constraint use the following syntax.

```
conforms body.
```

Definition 33 (Model Conformance)**.** A model conforms to its domain if:

– Every assertion in M is constructed from new-kind constructors.
– Every value in M is well-typed.
– The body of every conformance constraint is satisfied for some substitution of the variables.

Internally, FORMULA creates a special derived constant *D.conforms* for each domain D. All conformance constraints must be provable for *D.conforms* to be provable. Additionally, conformance constraints are introduced for the relation / function constraints appearing in type declarations. Table 7 lists the predefined derived constants. Users may write rules referring to these constants, but it is illegal to add a rule that proves them. Only the compiler can add rules proving these constants. We now list the complete domains for directed acyclic graphs and relational trees.

Example 6 (Directed Acyclic Graphs).

```
domain DAGs
{
    New-kind constructors.
    V ::= new (lbl: Integer).
    E ::= new (src: V, dst: V).
    Derived-kind constructors.
    path ::= (V, V).
    Computation of transitive closure.
    path(u, w) :- E(u, w); E(u, v), path(v, w).
    Acyclicity constraint.
    conforms no path(u,u).
}
```

Table 7. Table of predefined derived constants

Predefined Derived Constants

Name	Meaning
`D.conforms`	Provable if all conformance constraints are satisfied.
`D.notRelational`	Provable if a provable value contains $f(\ldots, t_i, \ldots)$, f is relational on position i, $t_i = g(\ldots)$, and $g(\ldots)$ is not provable.
`D.notFunctional`	Provable if a constructor is declared to be a (partial) function, but its provable values map an element from the domain of the function to several distinct elements in the codomain.
`D.notTotal`	Provable if a constructor is declared to be a total function, but some element of its domain is not mapped to an element of its codomain.
`D.notInjective`	Provable if a constructor is declared to be a (partial) injection, but several elements of its domain are mapped to the same element in its codomain.
`D.notInvTotal`	Provable if a constructor is declared to be a (partial) surjection, but there is an element of its codomain for which no element of the domain is mapped.

Example 7 (Relational Trees).

```
domain RelTrees_Final
{
    New-kind constructors.
    V        ::= new (lbl: Integer + String).
    Parent   ::= fun (chld: V => cxt: any {ROOT} + Context).
    Context ::= new (childPos: {LFT, RT}, prnt: V).
    Derived-kind constructors.
    anc ::= (ancestor: V, descendant: V).
    Computation of ancestors.
    anc(u, w) :- Parent(w, Context(_, u)); anc(u, v), anc(v, w).
    Computation of too-many children.
    tooManyChildren :-
       Parent(x, Context(pos, parent)),
       Parent(y, Context(pos, parent)),
       x != y.
    Conformance constraints
    conforms no anc(u, u).
    conforms no tooManyChildren.
}
```

3.9 Extracting Proofs

Often times it is useful to know *why* a query evaluates to true. If a query evaluates to true, then a *proof tree* can be obtained showing how the rules prove the query. In a new file type the code from Example 6 along with the code below:

```
model LittleCycle of DAGs
{
   v1 is V(1).
   v2 is V(2).
   E(v1, v2).
   E(v2, v1).
}
```

Execute the query (and observe that it evaluates to true):

```
[]> query LittleCycle path(u, u)
```

Next type:

```
[]> proof 0
```

(0 is the id of the query task; type the particular id of your task.) The following output is displayed:

```
1:  Truth value: true
2:
3:  _Query_263ea486_4485_4bff_bda4_c99ea8f94c27.requires :- (2, 1)
4:    ~dc0 equals
5:     _Query_263ea486_4485_4bff_bda4_c99ea8f94c27.~requires0 :- (2, 1)
6:       ~dc0 equals
7:        path(V(2), V(2)) :- (6, 4)
8:          ~dc0 equals
9:           E(V(2), V(1)) :- (14, 4)
10:          .
11:         ~dc1 equals
12:          path(V(1), V(2)) :- (6, 4)
13:            ~dc0 equals
14:             E(V(1), V(2)) :- (13, 4)
15:             .
16:          .
17:       .
18:    .
19: .
20:
21: Press 0 to stop, or 1 to continue
```

Line 1 indicates that the query evaluated to true. The remaining lines show a nested hierarchy of rules along with the values of the matching constraints that triggered the rule. The rules in lines 3 and 5 where generated by the compiler to hold the body of the query expression; they can be ignored. Line 7 shows

Table 8. Table of domain symbol placement

Placement of Domain Symbols	
Kind of Declaration	Symbols and Placement
`Type ::= ...`	The symbol `Type` and the *type constant* `#Type` are placed in the root. If declaration is an n-ary constructor declaration then type constants `#Type[0]`, ..., `#Type[n-1]` are also placed in the root.
`{ ..., Cnst, ... }`	The new-kind user constant `Cnst` is placed in the root.
`DerCnst :- ...`	The derived-kind user constant `DerCnst` is placed in the namespace D.
`x`	A variable introduced in a rule is placed in the root namespace. Variables can be introduced independently by many rules; this does not cause a conflict.
`D.Constant`	A predefined symbol defined to be the union of all new-kind constants (including numerics and strings). Placed in the namespace D along with `D.#Constant`.
`D.Data`	A predefined symbol defined to be the union of all new-kind constants (including numerics and strings) and data constructors. Placed in the namespace D along with `D.#Data`.
`D.Any`	A predefined symbol defined to be the union of all types in the domain. Placed in the namespace D along with `D.#Any`.

that a loop $path(V(2), V(2))$ was proved using the rule at line 6 column 4 in the domain definition (i.e. the transitive closure rule). This rule used the proofs of $E(V(2), V(1))$ (line 9) and $path(V(1), V(2))$ (line 12). The proof of $E(V(2), V(1))$ comes directly from the model assertion located at (14, 4). The proof of $path(V(1), V(2))$ requires another invocation of the transitive closure rule using the model assertion at (13, 4).

Press 1 key to obtain another proof of the query. Press the 1 key again and FORMULA exits the display loop because there are no more proofs to show. In fact, this statement is not entirely true; there are infinitely many proof trees that prove the loop, but most of them depend on a subproof of a loop and are not interesting. For instance, once $path(V(2), V(2))$ is proved, the the transitive closure rule can be applied again to obtain another proof for $path(V(2), V(2))$. FORMULA only shows minimal proof trees, and ignores (or *cuts*) proofs containing a subproof of the property.

The proof command is a bit more general. It can take a value (without variables) and return a truth value and possibly a proof tree. Try:

```
[]> proof 0 path(V(1), V(2))
```

Three results are possible:

– The truth value can be *true* and trees are displayed.
– The truth value can be *false* and no trees are displayed.
– The truth value can be *unknown*. This occurs if FORMULA did not evaluate enough rules to decide if the value is provable.

When a query operation is executed FORMULA will decide which rules are relevant to the query. It will report the truth value of *unknown* if the proof command is called with a value whose proof might require unexecuted rules. (This can also occur if the query execution was terminated prematurely.)

4 Domain and Model Composition

Domains and models can be combined to build up more complicated modules. Domain composition allows type declarations, rules, and conformance constraints to be combined. Model composition allows sets of assertions and aliases to be combined.

4.1 Namespaces

To understand composition, we must first discuss how symbols are organized. Every symbol s declared in a module is placed into a *namespace* n. The complete name of a symbol is $n.s$. We hinted at the existence of namespaces when describing derived constants such as *RelTrees.hasCycle* or *DAGs.conforms*. They also appeared in the symbolic constants *Gex'.%v1* and*Gex'.%e_1_2*. The *root namespace* has no name at all. The complete name of a symbol s placed in the root namespace is simply s. When modules are composed their namespaces are merged. Composition fails if the combined modules declare a symbol with the same full name but in semantically different ways.

Every domain D starts with two namespaces: the root namespace and a namespace D, which is a child of the root. Whether a declaration places symbols in the root or the D namespace depends on the kind of declaration. Table 8 describes the introduction and placement of domain symbols. Note that *type constants* are used to support reflection, but we have not discussed them yet.

Load the file containing Example 7 (*RelTrees_Final*) and type:

```
[]> det RelTrees_Final
```

You will see the complete set of symbols and their placement into namespaces. (Additional compiler generated symbols are listed as well.)

```
Symbol table
      Space      |       Name       | Arity | Kind
-----------------|------------------|-------|-------
                 |     Boolean      |   0   |  unn
                 |     Context      |   2   |  con
                 |      FALSE       |   0   | ncnst
                 |     Integer      |   0   |  unn
                 |       LFT        |   0   | ncnst
                 |     Natural      |   0   |  unn
                 |    NegInteger    |   0   |  unn
                 |      Parent      |   2   |  map
                 |    PosInteger    |   0   |  unn
                 |       ROOT       |   0   | ncnst
                 |        RT        |   0   | ncnst
                 |       Real       |   0   |  unn
                 |      String      |   0   |  unn
                 |       TRUE       |   0   | ncnst
                 |        V         |   1   |  con
                 |       anc        |   2   |  con
RelTrees_Final   |       Any        |   0   |  unn
RelTrees_Final   |     Constant     |   0   |  unn
RelTrees_Final   |       Data       |   0   |  unn
RelTrees_Final   |     conforms     |   0   | dcnst
RelTrees_Final   |  notFunctional   |   0   | dcnst
RelTrees_Final   |   notInjective   |   0   | dcnst
RelTrees_Final   |    notInvTotal   |   0   | dcnst
RelTrees_Final   |  notRelational   |   0   | dcnst
RelTrees_Final   |     notTotal     |   0   | dcnst
RelTrees_Final   | tooManyChildren  |   0   | dcnst
RelTrees_Final   |    ~conforms0    |   0   | dcnst
RelTrees_Final   |    ~conforms1    |   0   | dcnst

Type constants:  #Boolean #Context #Context[0] #Context[1]
                 #Integer #Natural #NegInteger #Parent #Parent[0]
                 #Parent[1] #PosInteger #Real #String #V #V[0]
                 #anc #anc[0] #anc[1] RelTrees_Final.#Any
                 RelTrees_Final.#Constant RelTrees_Final.#Data
Symbolic constants:
Rationals:
Strings:
Variables: parent pos u v w x y ~arg1 ~arg2 ~arg2'
```

A model M of domain D contains all the same definitions as D but adds an additional namespace called M under the root. All aliases are placed here as symbolic constants.

```
[]> det LittleCycle
```

```
Listing omitted
Symbolic constants:  LittleCycle.%v1 LittleCycle.%v2
Listing omitted
```

Altogether the *LittleCycle* model contains three namespaces: the root namespace and two sub-namespaces called *DAGs* and *LittleCycle*.

4.2 Domain Composition

Domains are composed by writing:

```
domain D includes D1, ..., Dn { ... }
```

Composition imports all the declarations of $D_1, \ldots, D_n$ into D. If a symbol is in namespace n in D_i, then it remains in namespace n in D. If this merging causes a symbol to receive contradictory declarations, then an error is reported. As a corollary, importing the same domain several times has no effect, because all declarations trivially agree. Here is an example of a problematic composition:

```
domain D1 {  q :- X = 0. }
domain D2 {  T ::= { X }. }
domain D includes D1, D2 { }
```

In domain D_1 the symbol X is a variable, but in domain D_2 it is a user constant. These declarations are incompatible and an error is returned:

```
ex.4ml (3, 23): The symbol X has multiple definitions.
```

Here is a more subtle example:

```
domain D1 {  List ::= new (Integer, any List + {NIL}). }
domain D2 {  List ::= new (Real, any List + {NIL}). }
domain D includes D1, D2 { }
```

Though both domains agree that $List(,)$ is a binary constructor; they disagree on the type constraints. However, this composition is legal:

```
domain D1
{  List ::= new (Integer, any List + {NIL}). }
domain D2
{
   List ::= new (NegInteger + {0} + PosInteger, any List + {NIL}).
}
domain D includes D1, D2
```

```
{
   List       ::= new (Integer, any ListOrNone).
   ListOrNone ::= List + {NIL}.
}
```

Though *List* has a syntactically different definitions in each domain, all domains semantically agree on the values accepted by the *List*(,) constructor.

The *includes* keyword merges domain declarations. However, it does not force the importing domain to satisfy all the domain constraints of the imported domains. In the previous examples D contains rules for computing D*.conforms*, D_1*.conforms*, and D_2*.conforms*. However, D*.conforms* need not reflect the conformance constraints of D_1 and D_2. (Relation / function constraints will be respected, because they occur on type declarations.) A composite domain automatically inherits conformance constraints if the *extends* keyword is used in place of *includes*:

```
domain D extends D1, ..., Dn { ... }
```

Consider that the set of all DAGs is a subset of the set of all directed graphs, and the set of trees is a subset of the set of DAGs. This chain of restrictions can be specified as follows:

Example 8 (Classes of graphs).

```
domain Digraphs
{
   V ::= new (lbl: Integer).
   E ::= new (src: V, dst: V).
}

domain DAGs extends Digraphs
{
   path ::= (V, V).
   path(u, w) :- E(u, w); E(u, v), path(v, w).
   conforms no path(u, u).
}

domain Trees extends DAGs
{
   conforms no { w | E(u, w), E(v, w), u != v }.
}
```

4.3 The Renaming Operator

Suppose we would like to build a domain representing two distinct graphs, i.e. with two distinct edge and vertex sets. One construction would be the following:

```
domain TwoDigraphs
{
   V1 ::= new (lbl: Integer).
   E1 ::= new (src: V1, dst: V1).

   V2 ::= new (lbl: Integer).
   E2 ::= new (src: V2, dst: V2).
}
```

The constructors $V_1()$ and $E_1(,)$ construct elements from the first graph and the constructors $V_2()$ and $E_2(,)$ construct elements from the second graph. A model would contain vertices and edges that could always be classified as belonging to either the first or second graph.

While this specification accomplishes the goal, it is not very satisfying. *TwoGraphs* contains two deep copies of the *Digraphs* domain, but the constructors have been renamed in an ad-hoc manner. FORMULA provides a methodological way to accomplish this same task: the *renaming operator* ('::').

```
rename::Module
```

The renaming operator creates a new module of the same kind with a namespace called *rename*. It copies all definitions from *module* under the *rename* namespace, and rewrites all copied declarations to reflect this renaming. The only symbols immune to this operation are new-kind constants, which remain at the root of the freshly created module. The renaming operator has many uses, though we only demonstrate a few uses now. The following domain describes the set of all pairs of isomorphic DAGs.

Example 9 (Isomorphic DAGs).

```
domain IsoDAGs extends Left::DAGs, Right::DAGs
{
   Iso ::= bij (Left.V => Right.V).

   conforms no { e | e is Left.E(u, w),
                     Iso(u, u'), Iso(w, w'),
                     no Right.E(u', w') }.
   conforms no { e | e is Right.E(u', w'),
                     Iso(u, u'), Iso(w, w'),
                     no Left.E(u, w) }.
}
```

This domain contains two copies of the *DAGs* domain under the renamings *Left* and *Right*. It contains two vertex constructors called *Left.V()* and *Right.V()* as well as two edge constructors called *Left.E(,)* and *Right.E(,)*. Because *IsoDAGs* extends the renamed domains, the left graph and the right graph must satisfy *Left.DAGs.conforms* and *Right.DAGs.conforms* respectively. In other words, the constraints on the renamed structures are preserved. Line 3 introduces a new bijection for witnessing the isomorphism between the left and the right vertices. Notice that the types *Left.V* and *Right.V* are immediately available for use. Finally, the two additional conformance constraints require the *Iso* bijection to relate the vertices such that *Iso* is a proper isomorphism. Here is a model of the domain:

```
model LittleIso of IsoDAGs
{
   v1L is Left.V(1).
   v2L is Left.V(2).
   v1R is Right.V(1).
   v2R is Right.V(2).

   Left.E(v1L, v2L).
   Right.E(v2R, v1R).

   Iso(v1L, v2R).
   Iso(v2L, v1R).
}
```

FORMULA provides several shortcuts to avoid fully qualifying symbols. First, a symbol only needs to be qualified until there is a unique shortest namespace containing the symbol. Suppose there are symbols $X.f()$ and $X.Y.Z.f()$. Then $f()$ will be resolved to $X.f()$ because this is the shortest namespace containing a symbol called $f()$. Also, $Y.f()$ will be resolved to $X.Y.Z.f()$ because this is the shortest namespace containing a qualifier Y and the symbol $f()$. Second, the resolved namespace of a constructor is applied to arguments of the constructor. If this resolution fails, then resolution restarts from the root namespace. Consider this example:

```
Left.E(V(1), V(2))
```

The outer constructor *Left*.$E(,)$ is resolved to be in the *Left* namespace. Next, the inner constructor $V()$ is encountered and name resolution looks for a unique shortest qualifier under the *Left* namespace. This succeeds and resolves as *Left*.$V()$. Thus, we have avoided writing the qualifier *Left* on the occurrences of $V()$. Of course, it is acceptable to fully qualify the inner constructors.

```
Left.E(Left.V(1), Left.V(2))
```

In this case, the symbol *Left.Left*.$V()$ does not exist and so name resolution restarts from the root and resolves to *Left*.$V()$.

5 Interpreted Functions

5.1 Arithmetic Functions and Identities

Table 9. Table of arithmetic functions (I)

Arithmetic Functions (I)		
Syntax	Side Constraints	Result
`-x`	`x : Real`	$-x$
`x + y`	`x : Real, y : Real`	$x + y$
`x - y`	`x : Real, y : Real`	$x - y$
`x * y`	`x : Real, y : Real`	$x \cdot y$
`x / y`	`x : Real, y : Real, y != 0.`	$\frac{x}{y}$
`x % y`	`x : Real, y : Real, y != 0.`	$0 \leq r < \|y\|$, such that $\exists q \in \mathbb{Z}.\ x = q \cdot y + r$.
`count({...})`	–	The number of elements in $\{\ldots\}$.
`gcd(x, y)`	`x : Integer, y : Integer`	$\overset{def}{=} \begin{cases} \|x\| \text{ if } y = 0, \\ gcd(y, \|x\| \% \|y\|). \end{cases}$
`gcdAll(x, {...})`	–	The gcd of all integer elements, or x if there are no such elements.
`lcm(x, y)`	`x : Integer, y : Integer`	$\overset{def}{=} \begin{cases} 0 \text{ if } \|x\| + \|y\| = 0, \\ \|x \cdot y\| / gcd(x, y). \end{cases}$
`lcmAll(x, {...})`	–	The lcm of all integer elements, or x if there are no such elements.
`max(x, y)`	–	x if $x \geq y$; otherwise y.
`maxAll(x, {...})`	–	The largest element of $\{\ldots\}$ in the order of values; x if $\{\ldots\}$ is empty.

Table 10. Table of arithmetic functions (II)

Arithmetic Functions (II)		
Syntax	Side Constraints	Result
`min(x, y)`	–	x if $x \leq y$; otherwise y.
`minAll(x, {...})`	–	The smallest element of $\{\ldots\}$ in the order of values; x if $\{\ldots\}$ is empty.
`prod(x, {...})`	–	$\overset{def}{=} \begin{cases} x \text{ if } \{\ldots\} \cap \mathbb{R} = \emptyset, \\ \prod\limits_{e \in \{\ldots\} \cap \mathbb{R}} e. \end{cases}$
`qtnt(x, y)`	`x : Real`, `y : Real`, `y != 0`.	$q \in \mathbb{Z}$, such that $\exists 0 \leq r < \|y\|.\ x = q \cdot y + r$.
`sign(x)`	`x : Real`	$\overset{def}{=} \begin{cases} -1 & \text{if } x < 0, \\ 0 & \text{if } x = 0, \\ 1 & \text{if } x > 0 \end{cases}$
`sum(x, {...})`	–	$\overset{def}{=} \begin{cases} x \text{ if } \{\ldots\} \cap \mathbb{R} = \emptyset, \\ \sum\limits_{e \in \{\ldots\} \cap \mathbb{R}} e. \end{cases}$

Table 11. Table of arithmetic identities (LHS-s are not built-in operations)

Arithmetic Identities		
Left-Hand Side		Right-Hand Side
$abs(x)$	$=$	$max(x, -x)$.
$ceiling(x)$	$=$	$-qtnt(-x, 1)$.
$floor(x)$	$=$	$qtnt(x, 1)$.

5.2 Boolean Functions

Table 12. Table of Boolean functions

Boolean Functions		
Syntax	Side Constraints	Result
`and(x, y)`	`x : Boolean`, `y : Boolean`	$x \wedge y$.
`andAll(x,` $\{\ldots\}$`)`	–	$\overset{def}{=} \begin{cases} x \text{ if } \{\ldots\} \cap \mathbb{B} = \emptyset, \\ \bigwedge_{e \in \{\ldots\} \cap \mathbb{B}} e. \end{cases}$
`impl(x, y)`	`x : Boolean`, `y : Boolean`	$\neg x \vee y$.
`not(x)`	`x : Boolean`	$\neg x$.
`or(x, y)`	`x : Boolean`, `y : Boolean`	$x \vee y$.
`orAll(x,` $\{\ldots\}$`)`	–	$\overset{def}{=} \begin{cases} x \text{ if } \{\ldots\} \cap \mathbb{B} = \emptyset, \\ \bigvee_{e \in \{\ldots\} \cap \mathbb{B}} e. \end{cases}$

5.3 String Functions

In the table above, ϵ is the empty string and $s[i]$ is the single-character string at position i in s, for $0 \leq i < strLength(s)$.

Table 13. Table of string functions

String Operations		
Syntax	Side Constraints	Result
`isSubstring(x, y)`	`x : String`, `y : String`.	`TRUE` if x is a substring of y; `FALSE` otherwise. The empty string is only a substring of itself.
`strAfter(x, y)`	`x : String`, `y : Natural`.	Returns the largest substring starting at position y, or ϵ if $y \geq strLength(x)$.
`strBefore(x, y)`	`x : String`, `y : Natural`.	Returns the largest substring ending before position y, or ϵ if $y = 0$.
`strFind(x, y, z)`	`x : String`, `y : String`.	Returns the index of the first occurrence of x in y; z if y never appears.
`strFindAll(x, y, z, w)`	x is a w-terminated natural-list type constant `#F`. `y : String`, `z : String`.	Returns a w-terminated F'-list of all the indices where y occurs in z; w if it never occurs.
`strGetAt(x, y)`	`x : String`, `y : Natural`.	$\overset{def}{=} \begin{cases} x[y] \text{ if } y < strLength(x), \\ \epsilon \text{ otherwise.} \end{cases}$.
`strJoin(x, y)`	`x : String`, `y : String`.	$\overset{def}{=} \begin{cases} y \text{ if } x = \epsilon, \\ x \text{ if } y = \epsilon, \\ xy \text{ otherwise.} \end{cases}$.
`strLength(x)`	`x : String`.	Returns the length of x.
`strLower(x)`	`x : String`.	Returns the all-lower-case version of x.
`strReverse(x)`	`x : String`.	Returns the reverse of x.
`strUpper(x)`	`x : String`.	Returns the all-upper-case version of x.

5.4 List Functions

A *list constructor* is a constructor `F ::= (T0, T1)` such that `F` is a subtype of `T1`. A *T-list constructor* is a list constructor such that `T` is a subtype of `T0`. A *list type constant* `#F` is a type constant such that `F` is a list constructor. A list is *flat* if it has n elements placed as follows:

$$F(t_0, F(t_1, \ldots, F(t_{n-1}, w) \ldots))$$

The value w is called the *terminator*. In the table below all operations, except for `isSubTerm` and `lstFlatten`, assume flat lists.

Table 14. Table of list functions

List Functions		
Syntax	Side Constraints	Result
`isSubterm(x, y)`	–	`TRUE` if x is a subterm of y; `FALSE` otherwise. .
`lstAfter(x, y, z, w)`	x is a w-terminated list type constant `#F`, `z : Natural`.	y if $y \neq F(\ldots)$; w if $z \geq lstLength(y)$; a w-terminated F-list of all the elements at and after z.
`lstBefore(x, y, z, w)`	x is a w-terminated list type constant `#F`, `z : Natural`.	y if $y \neq F(\ldots)$; w if $z \leq 0$; a w-terminated F-list of all the elements before z.
`lstFind(x, y, z, w)`	x is a list type constant `#F`.	The first place where z occurs in the F-list y; w if z never occurs.
`lstFindAll(x, x', y, z, w)`	x is a list type constant `#F`, x' is a w-terminated natural-list type constant `#F'`.	Returns a w-terminated F'-list of all the indices where z occurs in y; w if it never occurs.
`lstFlatten(x, y, w)`	x is a w-terminated list type constant `#F`.	Converts y into w-terminated flat form if $y = F(\ldots)$; y otherwise.
`lstGetAt(x, y, z)`	`z : Natural`, `z < lstLength(x, y)`.	$\overset{def}{=} \begin{cases} h \text{ if } lstGetAt(x, F(h,t), 0), \\ lstGetAt(x, t, z-1). \end{cases}$
`lstLength(x, y)`	x is a list type constant `#F`.	$\overset{def}{=} \begin{cases} 0 \text{ if } y \neq F(h,t), \\ 1 + lstLength(t). \end{cases}$
`lstReverse(x, y)`	x is a list type constant `#F`.	y if $y \neq F(\ldots)$; otherwise reverses the list reusing the same terminator

5.5 Coercion Functions

In the table to follow, a *list type constant* `#F` is a type constant such that `F ::= (T1, T2)` and `F <: T2`.

Table 15. Table of coercion functions

Coercion Functions		
Syntax	Side Constraints	Result
`toList(x, y, {...})`	x is a list type constant, y : T2.	$F(t_1, \ldots, F(t_n, y) \ldots)$ where t_i are the sorted elements of $\{\ldots\}$ accepted by $T1$. Or y if no element is accepted.
`toNatural(x)`	–	Returns a unique natural for the value x.
`toString(x)`	–	Returns a unique string for the value x.
`toSymbol(x)`	–	Returns x if x is a constant; `#F` for $x = F(\ldots)$.

5.6 Reflection Functions

Table 16. Table of reflection functions

Reflection Functions		
Syntax	Side Constraints	Result
`rflGetArgType(x, y)`	x is a type constant `#F`, `y : Natural`, `y < rflGetArity(x)`.	Returns `#F[y]`, where `F ::= (T0,...,T`$_{n-1}$`)`
`rflGetArity(x)`	x is a type constant `#X`.	Returns n if `X ::= (T0,...,T`$_{n-1}$`)`; 0 otherwise.
`rflIsMember(x, y)`	y is a type constant `#Y`.	`TRUE` if x is a member of Y; `FALSE` otherwise.
`rflIsSubtype(x, y)`	x is a type constant `#X`, y is a type constant `#Y`.	`TRUE` if X is a subtype of Y; `FALSE` otherwise.