2. FORMULA: Domains and Models

Ethan Jackson, Nikolaj Bjørner and Wolfram Schulte
Research in Software Engineering (RiSE), Microsoft Research

Dirk Seifert, Markus Dahlweid and Thomas Santen
European Microsoft Innovation Center (EMIC), Microsoft Research

FORMULA

Modeling Foundations.

i. Type Declarations

http://research.microsoft.com/formula
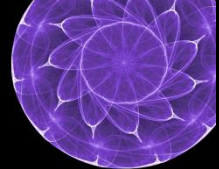
FORMULA

Modeling Foundations.

Domains Encapsulate Abstractions

$G \stackrel{\text{def}}{=} (V, E)$ such that $E \subseteq V \times V$.

```
domain Digraph
{
    //// ADT Declarations
    //// CLP rules/queries
}
```

Declares both an n-ary constructor and type with the same name.

$$con ::= (lbl_1 : \tau_1, \ldots, lbl_n : \tau_n).$$

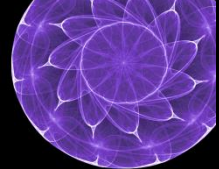Body of declaration gives type constraints and labels for each argument.

```
V    ::= (id: Integer).
E    ::= (src: V, dst: V).
path ::= (beg: V, end: V).
```

Base types are order-sorted and include constants and:

NegInteger, PosInteger, Natural, Integer, Real,
String, Boolean, Basic, Any.

The type of a constant is itself (denoting a singleton set containing the constant).

Declares a type naming a finite set of constants.

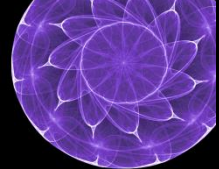$$enm ::= \{ c_1, ..., c_n \}.$$

Body contains constants.

```
Colors ::= { red, green, blue }.
Mixed  ::= { 1, 2.5, true, "hello" }.
```

Enumerations can introduce new constants, like red. Numeric and string constants are predefined, as are the distinguished constants true/false.
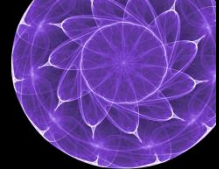
Declares a type naming a union of terms.

$$unn ::= \tau_1 + \cdots + \tau_n.$$

Union types are not tagged and equivalent to the mathematical union.

```
GraphThing    ::= V + E.
VertexOrColor ::= V + Colors.
```

Type Declarations – Recursive ADTs

Have enough ingredients to represent many structures.

```
Nil   ::= { nil }.
VList ::= Nil + VLCon.
VLCon ::= (head: V, tail: VList).
```

Unions can simulated multiple-inheritance and interfaces.

```
Colors ::= { red, green, blue }.
V      ::= (id : Integer).
ColV   ::= (id : Integer, col: Colors).
IV     ::= V + ColV.
E      ::= (src: IV, dst: IV).
```

Types are subtype related if one denotes a subset of the other.

$$\tau' <: \tau \ \stackrel{\text{def}}{=} \ [\![\tau']\!] \subseteq [\![\tau]\!]$$

Where $[\![\tau]\!]$ is the set of terms accepted by τ.

```
1 <: 1 ∪ 2 <: PosInteger <: Natural
    <: Integer <: Real <: Basic


  V <: GraphThing, V <: VertexOrColor,
GraphThing <: Any, VertexOrColor <: Any


    VLCon(V(Natural), nil) <: VList
```

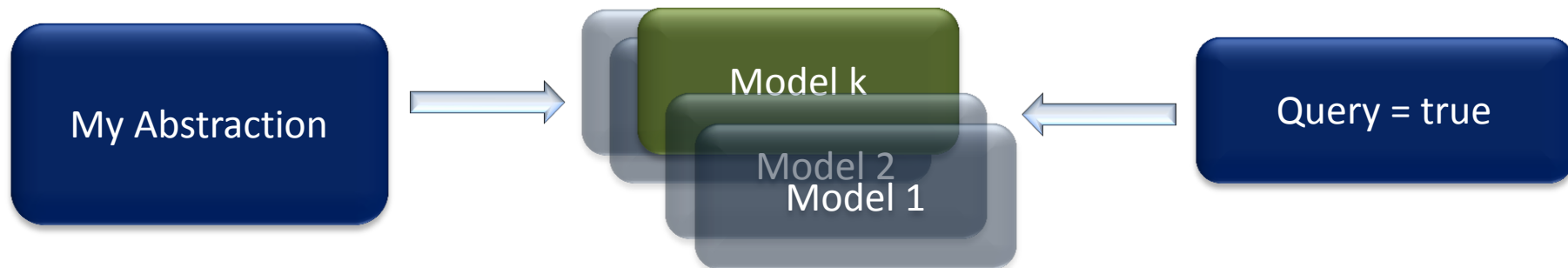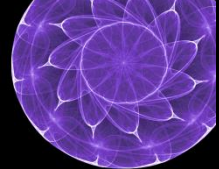Some constructors represent the building blocks of an abstraction. Others are used to store information derived from these building blocks.

```
domain Digraph
{
    primitive V ::= (id: Integer).
    primitive E ::= (src: V, dst: V).
    path ::= (beg: V, end: V).
}
```

The primitive keyword indicates the building blocks and impacts modeling finding, because models can only contain primitives.
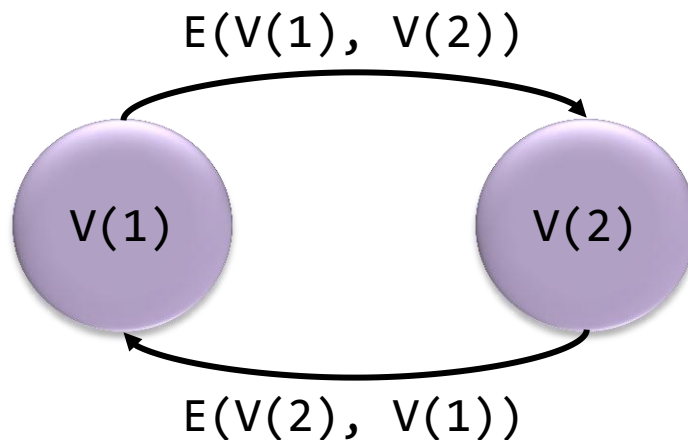
…that conform to a domain.

```
model G1 of Digraph
{
    V(1) V(2)
    E(V(1), V(2))
    E(V(2), V(1))
}
```


E(V(1), V(2))

V(1)   V(2)

E(V(2), V(1))
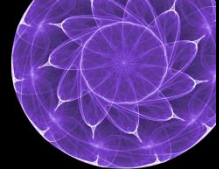
Can alias terms with the "is" keyword.

```
model G1p of Digraph
{
    v1 is V(1) v2 is V(2)
    E(v1, v2)
    E(v2, v2)
}
```
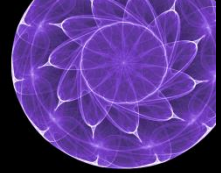
Compiler Checks Conformance

Compiler rejects this model, because it doesn't respect data type declarations.

```
✖ model G2 of Digraph
  {
      V(E(V(1), V(2)))
  }
```

Compiler rejects this model, because it contains a term that is not primitive:

```
✖ model G3 of Digraph
  {
      path(V(1), V(2))
  }
```

We will see more complex conformance constraints soon…

Models extend a domain with facts, and the result is a closed logic program.

```
model G1p of Digraph
{
    v1 is V(1) v2 is V(2)
    E(v1, v2) E(v2, v2)
}
```

$$\equiv$$

```
Digraph ∪

{  V(1). V(2). E(V(1), V(2)). E(V(2), V(2)). }
```

ii. Rules

http://research.microsoft.com/formula

For more complex constraints we need logic programs.

$$f(\vec{t}) \leftarrow f_1(\overrightarrow{t_1}), \dots, f_m(\overrightarrow{t_m}),$$
$$\boldsymbol{fail}\ g_1(\overrightarrow{s_1}), \dots, \boldsymbol{fail}\ g_n(\overrightarrow{s_n}),$$
$$c_1 rel_1\ c_1', \dots, c_o\ rel_o\ c_o'.$$

Rule construct new facts from the set of currently known facts K.
Roughly, a rule creates new knowledge for every substitution θ such that:

(1) Find predicates hold: $f_i(\overrightarrow{t_i}\theta) \in K$.
(2) Fail predicates hold: $g_i(\overrightarrow{s_i}\theta) \notin K$.
(3) Constraint predicates hold: $(c_i\theta, c_i'\theta) \in rel_i$.

$$rel_i \in \{\ =,\ !=,\ <,\ <=,\ >,\ >=\ \}$$

Order of rules/predicates doesn't matter.
Finds occur in higher quantifier scope than fails.

The following sets of rules are equivalent ways of defining the paths in a graph.

```
path(x, y) :- E(x, y).
path(x, z) :- path(x, y), path(y, z).


path(x, z) :- path(x, y), path(y, z).
path(x, y) :- E(x, y).


path(x, y) :- E(x, y).
path(x, z) :- E(x, y), path(y, z).


path(x, z) :- E(x, y), path(y, z).
path(x, y) :- E(x, y).
```
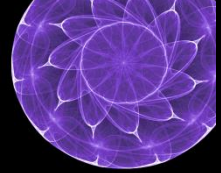
```
domain Digraph
{
    primitive V ::= (id: Integer).
    primitive E ::= (src: V, dst: V).
    path ::= (beg: V, end: V).

    path(x, y) :- E(x, y).
    path(x, z) :- path(x, y), path(y, z).
}
```

Knowledge

The knowledge generated by a model is the least K^∞ (least fixpoint) where no rules extend K^∞.

$$K_0 = \{ V(1), V(2), E(V(1), V(2)), E(V(2), V(2)) \}$$

▶ `path(x, y) :- E(x, y).`

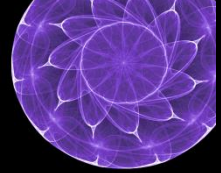Let $\theta(x) \mapsto V(1), \theta(y) \mapsto V(2)$ then $E(x\theta, y\theta) \in K_0$

$$\{ path(x\theta, y\theta) \stackrel{\text{def}}{=} path(V(1), V(2)) \} \cup K_0 = K_1$$

Let $\theta(x) \mapsto V(2), \theta(y) \mapsto V(2)$ then $E(x\theta, y\theta) \in K_1$

$$\{ path(x\theta, y\theta) \stackrel{\text{def}}{=} path(V(2), V(2)) \} \cup K_1 = K_2$$

$$K_2 = K_0 \cup \{ path(V(1), V(2)), path(V(2), V(2)) \}$$

*FORMULA computes the lfp bottom-up, like datalog instead of prolog.

▶ `path(x, y) :- path(x, y), path(y, z).`

Let $\theta(x) \mapsto V(1), \theta(y) \mapsto V(2)$ then $\text{path}(x\theta, y\theta) \in K_2$
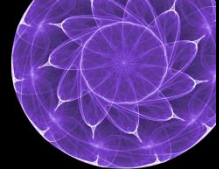
$$\{ path(x\theta, y\theta) \stackrel{\text{def}}{=} path(V(1), V(2)) \} \cup K_2 = K_3$$

Let $\theta(x) \mapsto V(2), \theta(y) \mapsto V(2)$ then $\text{path}(x\theta, y\theta) \in K_3$

$$\{ path(x\theta, y\theta) \stackrel{\text{def}}{=} path(V(2), V(2)) \} \cup K_3 = K_4$$

$$K_4 = K_1 \cup \{ path(V(1), V(2)), path(V(2), V(2)) \} = K_1$$

Therefore $K^\infty \stackrel{\text{def}}{=} K_1$ is the lfp, which is everything knowable from the model.

Rules can only do a finite amount of work at each step.
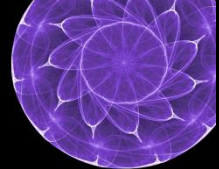
Rule: `h :- B[K].`

Head   Body, a first order formula
with free variable K

Let $sat(B, K)$ be the set of all satisfying substitutions for B given K.

Range restriction.

If $|K|$ is finite then $|sat(B, K)|$ must be finite.

*Range restriction allows for an effective symbolic execution procedure to eliminate nested quantifiers and fixpoints.

Range restriction must be approximated.
(Some range restricted programs are rejected.)

❌ `path(x, y) :- V(x).`

▲

Derives an infinite number of paths.
Rule: Head variables must be bound on RHS.

❌ `path(x, y) :- V(x), fail V(y).`

▲
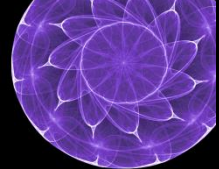
Derives an infinite number of paths.
Rule: Head variables must appear outside
of fail scopes.

❌ `path(x, y) :- V(x), y + 1 = x*x.`

▲

Range restricted, but cannot prove it.
Rule: Write directed equalities to make this obvious.

These programs are range restricted.

✓ `path(x, V(z)) :- E(x, y), V(z) = y.`
▲

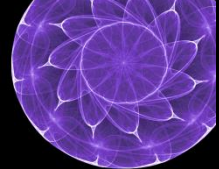The compiler knows z is a subterm of y and y is range restricted because it is a subterm of $E(x, y) \in K$.

✓ `path(x, y) :- V(x), y = x*x - 1.`
▲

Because y is function of x, which is range restricted.

Rules must construct well-typed terms regardless of the model used to close the program.
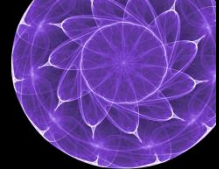
Rule: `h :- B[K].`
And: $x \in$ `B`

Then an over-approximation of the values x can take is:

$$values(x) \overset{\text{def}}{=} \bigcup_K \{ \theta(x) | \theta \in sat(B, K)\}$$

Type inference assigns x a type τ_x which over approximates $values(x)$ by a regular set:

$$values(x) \subseteq [\![\tau_x]\!]$$

A rule is type safe if the inferred type of the head is a subtype of the declared type.

$$f(\vec{t}[\vec{x}]) \; \texttt{:-} \; B\texttt{.}$$

The inferred type of the head is:

$$f(\vec{t}[x_i \backslash \tau_{x_i}])$$

The inferred type of the head must satisfy:

Type safety.

$$f(\vec{t}[x_i \backslash \tau_{x_i}]) \; \texttt{<:} \; f$$

A rule is also rejected if the body cannot be satisfied:

$t : \perp$ for some term in B.
$p : false$ for some predicate in B.

The following rules are not type safe

❌ `path(x, V(y)) :- V(x), y = x / 2.`
▲

A world with an odd numbered vertex gives y a
fractional value, so `y : Real`. Therefore
`path(x, V(y)) : path(V, V(Real))`.

❌ `path(x, y) :- E(x, V(y)), x = y.`
▲

Bug here, because `x : V` and `y : Integer`. But
`x = y` implies `x, y : V ∩ Integer ≡ ⊥`.

❌ `path(V(x), p) :- V(x), V(y), p = path(V(x), V(y)).`
▲

Because `path(x, p) : path(V, path)`.

These rules are type safe.

```
Colors ::= { red, green, blue }.
V      ::= (id : Integer).
ColV   ::= (id : Integer, col: Colors).
IV     ::= V + ColV.
E      ::= (src: IV, dst: IV).
cpath  ::= (beg: ColV, end: ColV).
```

✔ cpath(x, y) :- E(x, y), x.col = y.col.
              ▲

Because only colored vertices have col arguments.
cpath(x, y) : cpath(ColV, ColV).

✔ cpath(x, y) :- E(x, y), x.col = y.col, x.col = red.
              ▲

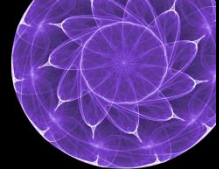cpath(x, y) : cpath(ColV(Integer, red), ColV(Integer, red)).

iii. Queries

http://research.microsoft.com/formula

FORMULA

Modeling Foundations.

Query definitions are special rules where the LHS is a constant.
Query constants are separate from data type constants.

$$G \overset{\text{def}}{=} (V, E) \text{ such that } E \subseteq V \times V.$$

▲

How do we encode this constraint?

```
undeclVertex := E(V(x), _), fail V(x).
```
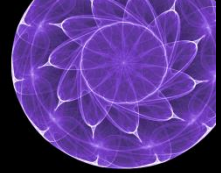
▲　　　　　▲　　　　　▲

Query name　　Query operator　　Fresh "don't care" variable

```
undeclVertex := E(_, V(y)), fail V(y).
```

Multiple query definitions are allowed;
they permit the same query to be derived in several different ways.

```
model GUndecl of Digraph
{ V(1), E(V(1), V(10)) }
```
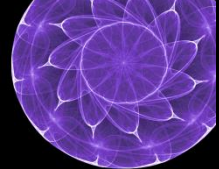
$$K_0 = \big\{ V(1), E\big(V(1), V(10)\big)\big\}$$

▶ `undeclVertex := E(_, V(y)), fail V(y).`

Let $\theta(\_) \mapsto V(1), \theta(y) \mapsto 10$ then $\mathsf{E}(\_\theta, V(y\theta)) \in K_0$ and $V(y\theta) \notin K_0$.
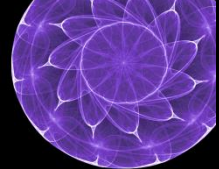
$$\mathsf{undeclVertex} \in K_1$$

We must tell the compiler that undeclared vertices are disallowed.
This is done using the special `conforms` query.

```
conforms := !undeclVertex.
```
▲
Queries are treated like Boolean variables on the RHS.

Every domain has a `conforms` query, even if it isn't explicitly defined.
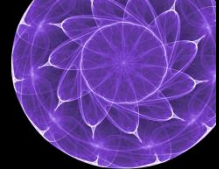All conformance constraints must appear in the `conforms` query.

```
domain Digraph
{
    primitive V ::= (id: Integer).
    primitive E ::= (src: V, dst: V).
    path ::= (beg: V, end: V).

    path(x, y) :- E(x, y).
    path(x, z) :- path(x, y), path(y, z).

    undeclVertex := E(V(x), _), fail V(x).
    undeclVertex := E(_, V(y)), fail V(y).
    conforms := !undeclVertex.
}
```

In addition to range restriction and well-typedness, FORMULA must be terminating.

Termination allows for a bottom-up fixpoint procedure and generalization to symbolic execution.

Termination

The knowledge generated by an arbitrary model must be finite,
i.e. a finite fixpoint must exist.

❌ `path(V(y), V(y)) :- path(V(x), _), y = x + 1.`

*For those familiar with backwards-chaining LP, this requires an adjustment. Currently, FORMULA does not perform any termination analysis.

Not all logic programs have unique interpretations.
Stratification is a syntactic constraint on `fail`, which guarantees unique interpretations.

FORMULA programs must be stratified. This means specifications are in a fragment of FPL where model finding is easier to encode into constraint solvers.

Stratification

There is no subset of rules producing an f-term by failing on an f-term.

*The compiler checks stratification.

This specification is not stratified.

```
domain DigraphNotStrat
{
    primitive V ::= (id: Integer).
    primitive E ::= (src: V, dst: V).
    path ::= (beg: V, end: V).

    path(x, y) :- E(x, y).
 ✖ path(x, x) :- path(x, y), fail path(y, y).
}

model Problem of DigraphNotStrat
{
    V(1) V(2)
    E(V(1), V(2))
    E(V(2), V(1))
}
```

Consider computing the knowledge of `Problem`.

$$K_2 = \{ \ldots, path(V(1), V(2)), path(V(2), V(1)) \}$$

▶ `path(x, x) :- path(x, y),` `fail` `path(y, y).`

Let $\theta(x) \mapsto V(1), \theta(y) \mapsto V(2)$ then
$path(x\theta, y\theta) \in K_2$ and $path(y\theta, y\theta) \notin K_2$

$$\{ path(x\theta, x\theta) \stackrel{\text{def}}{=} path(V(1), V(1)) \} \cup K_2 = K_3$$

Let $\theta(x) \mapsto V(2), \theta(y) \mapsto V(1)$ then
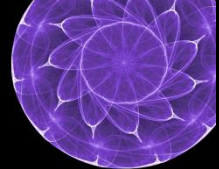$path(x\theta, y\theta) \in K_3$ and $path(y\theta, y\theta) \in K_3$ ✖

But, if the second substitution had been chosen first, then the result would be different. There are two least fixpoints:

$$K_3 = K_2 \cup \{ path(V(1), V(1)) \} \qquad K'_3 = K_2 \cup \{ path(V(2), V(2)) \}$$

vi. More Language Details

http://research.microsoft.com/formula

Shortcuts - Closed

Constraints, such as no undeclared vertices, are very common. FORMULA provides annotations that automatically introduce the appropriate rules and modify `conforms`.

The closed annotation; subterms must be known (in K^∞).

$$[Closed(lbl_{d_1}, \dots, lbl_{d_m})]con ::= (lbl_1 : \tau_1, \dots, lbl_n : \tau_n).$$

The constraint is trivially satisfied for constants, because all constants are known.

Unnecessary constraint

```
▶ [Closed(id)]
  primitive V ::= (id: Integer).
  [Closed(src, dst)]
  primitive E ::= (src: V, dst: V).
```

The unique annotation; any two con-terms with the same arguments on $\{d_1, ..., d_m\}$ must have the same arguments on $\{e_1, ..., e_o\}$.

$$\left[Unique\left(lbl_{d_1}, ..., lbl_{d_m} \rightarrow lbl_{e_1}, ..., lbl_{e_o}\right)\right]$$
$$con ::= (lbl_1 : \tau_1, ..., lbl_n : \tau_n).$$

For colored vertices there is a unique color for every vertex id:

```
[Unique(id -> col)]
ColV    ::= (id : Integer, col: Colors).


model GBad of ColDigraph
  {
      ColV(1, red)
      ColV(1, blue)
  }
```

The total annotation; there must be a con-term for all possible combinations of legal arguments $\{d_1, \dots, d_m\}$.

$$\left[Total\left(lbl_{d_1}, \dots, lbl_{d_m}\right)\right]$$
$$con ::= (lbl_1 : \tau_1, \dots, lbl_n : \tau_n).$$

This means there must be a vertex term for every possible integer.

```
        [Total(id)]
   ✖ primitive V ::= (id: Integer).
```

This use of the annotation is illegal, because a model cannot have an infinite number of terms.

If Total is used with Closed, then the compiler knows only a finite number of elements are accepted.

```
primitive V ::= (id: Integer).
[Total(src, dst)][Closed(src, dst)]
primitive E ::= (src: V, dst: V).
```

✔
```
model GGood of Digraph
{
    v is V(1) v is V(2)
    E(v1, v1) E(v1, v2)
    E(v2, v1) E(v2, v2)
}
```

✖
```
model GBad of Digraph
{
    v is V(1) v is V(2)
    E(v1, v1) E(v1, v2)
    E(v2, v1)
}
```

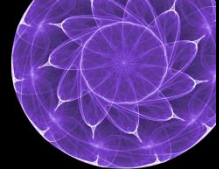These annotations form the building blocks for other common constraints.

$$\left[Function\left(lbl_{d_1}, \ldots, lbl_{d_m} \rightarrow lbl_{e_1}, \ldots, lbl_{e_o}\right)\right]$$
$$con ::= (lbl_1 : \tau_1, \ldots, lbl_n : \tau_n).$$

$$\left[Unique\left(lbl_{d_1}, \ldots, lbl_{d_m} \rightarrow lbl_{e_1}, \ldots, lbl_{e_o}\right)\right]$$
$$\left[Total\left(lbl_{d_1}, \ldots, lbl_{d_m}\right)\right]$$
$$con ::= (lbl_1 : \tau_1, \ldots, lbl_n : \tau_n).$$

$$\left[Injection\left(lbl_{d_1}, \dots, lbl_{d_m} \rightarrow lbl_{e_1}, \dots, lbl_{e_o}\right)\right]$$
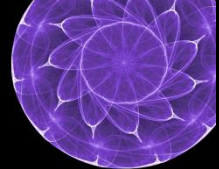$$con ::= (lbl_1 : \tau_1, \dots, lbl_n : \tau_n).$$

$$\left[Function\left(lbl_{d_1}, \dots, lbl_{d_m} \rightarrow lbl_{e_1}, \dots, lbl_{e_o}\right)\right]$$
$$\left[Unique\left(lbl_{e_1}, \dots, lbl_{e_o} \rightarrow lbl_{d_1}, \dots, lbl_{d_m}\right)\right]$$
$$con ::= (lbl_1 : \tau_1, \dots, lbl_n : \tau_n).$$

$$\left[Surjection\left(lbl_{d_1}, \dots, lbl_{d_m} \to lbl_{e_1}, \dots, lbl_{e_o}\right)\right]$$
$$con ::= (lbl_1 : \tau_1, \dots, lbl_n : \tau_n).$$

$$\left[Function\left(lbl_{d_1}, \dots, lbl_{d_m} \to lbl_{e_1}, \dots, lbl_{e_o}\right)\right]$$
$$\left[Total\left(lbl_{e_1}, \dots, lbl_{e_o}\right)\right]$$
$$con ::= (lbl_1 : \tau_1, \dots, lbl_n : \tau_n).$$

$$\left[Bijection\left(lbl_{d_1}, \dots, lbl_{d_m} \rightarrow lbl_{e_1}, \dots, lbl_{e_o}\right)\right]$$
$$con ::= (lbl_1 : \tau_1, \dots, lbl_n : \tau_n).$$

$$\left[Injection\left(lbl_{d_1}, \dots, lbl_{d_m} \rightarrow lbl_{e_1}, \dots, lbl_{e_o}\right)\right]$$
$$\left[Injection\left(lbl_{e_1}, \dots, lbl_{e_o} \rightarrow lbl_{d_1}, \dots, lbl_{d_m}\right)\right]$$
$$con ::= (lbl_1 : \tau_1, \dots, lbl_n : \tau_n).$$

```
domain Digraph
{
    primitive V ::= (id: Integer).
    [Closed(src, dst)]
    primitive E ::= (src: V, dst: V).
    path ::= (beg: V, end: V).

    path(x, y) :- E(x, y).
    path(x, z) :- path(x, y), path(y, z).
}
```

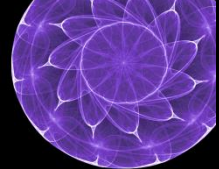Multiple rules with the same head can be written as one rule with comma-separated heads.

```
path(x, y), path(y, x) :- E(x, y).
```

▲

Computes symmetric
closure of E.

Multiple rules with the same and different bodies can be written as one rule with semi-colon-separated bodies.

```
path(x, z) :- E(x, z); path(x, y), path (y, z).
```

▲

Each semicolon is a
new scope

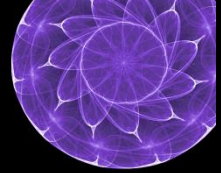The FORMULA syntax also supports some generalizations of classic logic programming.

```
GraphThing ::= V + E.

threePath := x is GraphThing, y is GraphThing,
             x.dst = y.src, x.src != y.dst.
```

The variables `x` and `y` range over all known terms conforming to the type `GraphThing`

```
SameCol(v1, v2) := v1 is ColV(_,col), v2 is ColV(_, col).
```

Can also use `is` to bind a variables to find/fail predicates.

Attaches a type membership constraint to a variable.

```
Mixed  ::= { 1, 2.5, true, "hello" }.

v1 := V(x), x in Mixed.
```

$$\text{Integer} \cap \text{Mixed} \equiv 1$$

Queries can be used only on the RHS of other query definitions.
When this occurs, they are combined using the usual Boolean connectives instead of LP.

```
q := !(q1 | q2) & q3.
```

        ▲   ▲   ▲

       not   or   and

Query definitions should be oriented, i.e. no circular-definitions.

```
✖ q1 := !q2. q2 := !q1.
```

vi. Partial Models

http://research.microsoft.com/formula

FORMULA

Modeling Foundations.

Partial models allow targeted model finding.
They are essentially lower bounds on the sort of models we want to find.

```
partial model P of Digraph
{
    E(_, V(3)) E(x, y) E(y, z)
}       ▲                 ▲
    fresh variable      named variable
```

The model finder must return a solution containing these terms under some substitution.

The model finder tries to assign consistent cardinalities to primitives so that the Closed, Total, and Unique annotations can be satisfied.

$$\#E > 0 \Rightarrow \#V > 0.$$

This problem is undecidable (non-linear Diophantine inequalities), but solved up-to some maximum cardinalities and greedily minimized.

Added two
vertices with ▶
unknown IDs.

```
partial model P of Digraph
{
    V(_) V(_)
    E(_, V(3)) E(x, y) E(y, z)
}
```

The partial model is automatically extended with consistent number of non-ground terms.

The program is symbolically executed under the extended partial model and the target query is solved using Z3.

```
partial model P of Digraph
{
    V(_) V(_)
    E(_, V(3)) E(x, y) E(y, z)
}
```
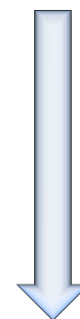
$$\theta(\_0) \mapsto -78$$
$$\theta(\_1) \mapsto 3$$
$$\theta(x), \theta(y), \theta(z) \mapsto V(3)$$
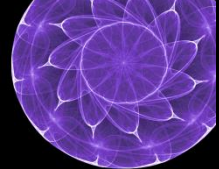
```
model P_1 of Digraph at "../ICTAC.4ml"
{
    V(-78)
    V(3)
    E(V(3),V(3))
}
```

Each solution provides a substitution that is used to convert a partial model into a complete model.

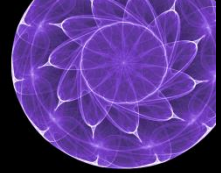There are two modes of searching for multiple solutions: **point exclusion** and **symmetry exclusion**.

Point exclusion prevents the solver from returning any solution that has already been seen, and then asks for another one.
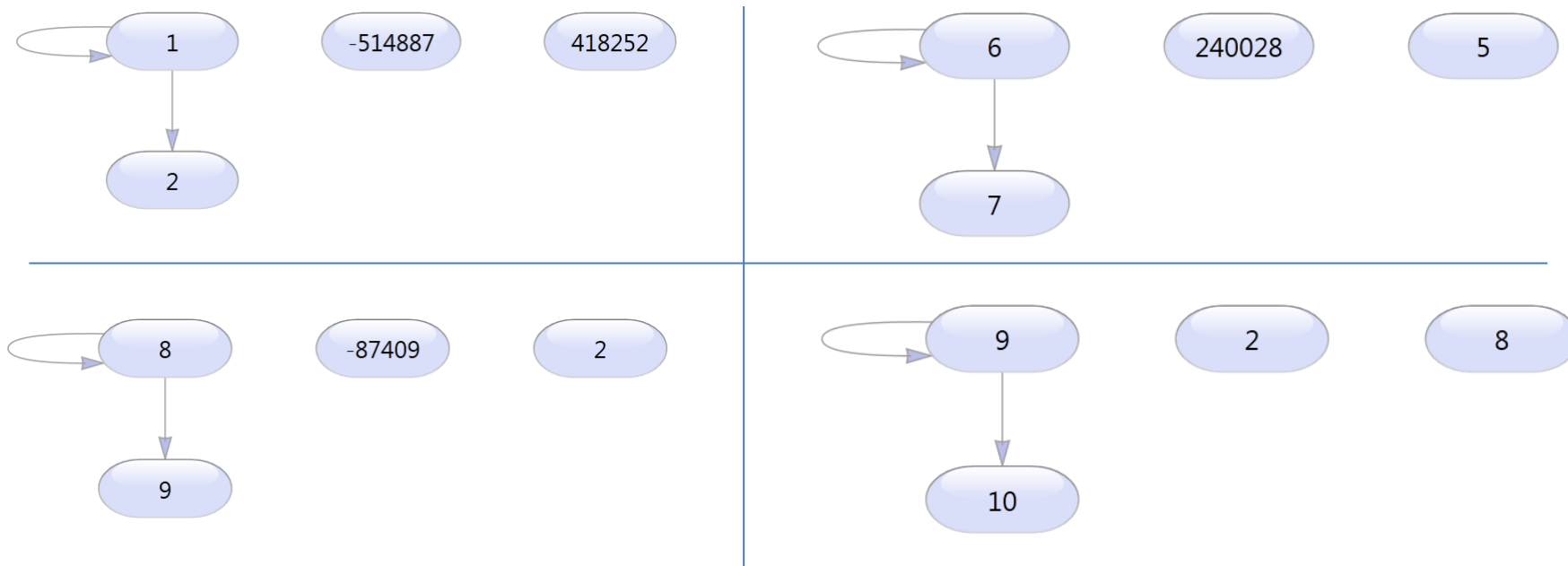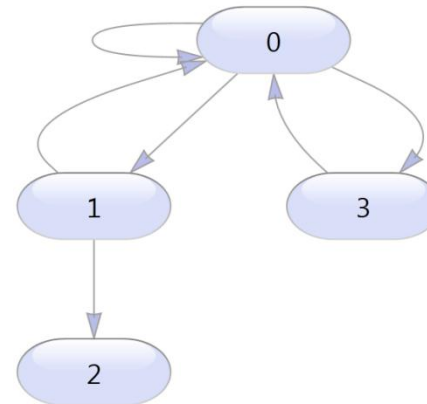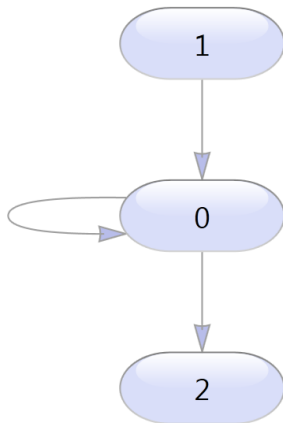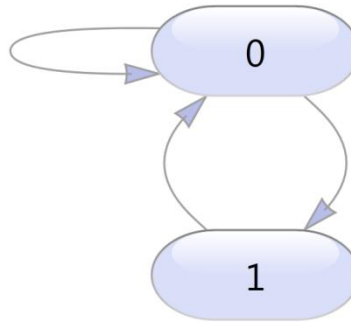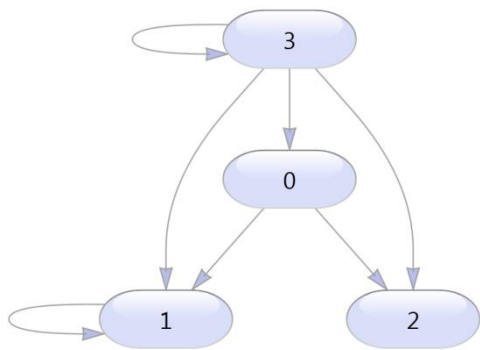

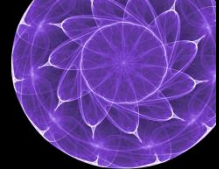Without additional configuration, point exclusion is the default search mechanism.

Symmetry exclusion presents the solver from returning isomorphic solutions, and also externally randomizes the search for non-isomorphic solutions.

```
[Config(EXPLORATION_METHOD=SymmetryExclusion)]
partial model P3 of Graph { … }
```

There are several annotations for controlling the size and contents of the partial model.

Add n terms of the form `f(_,...,_)` to the partial model

$$[Introduce(f, n)]$$

```
[Introduce(E, 10)]
partial model P2 of Digraph { }
```

The solution must contain at least m distinct f-terms and at most n distinct f-terms

$$[Cardinality(f, m, n)]$$

The solution must contain at least m distinct f-terms

$$[Cardinality(f, m)]$$

```
[Cardinality(E,5,10)]
partial model P2 of Digraph { }
```

By default, the model finder does not automatically increase the size of the partial model after the first cardinality inference. However, this can easily be adjusted.

The model finder will now continue to increase the size of the partial model. At each step a requested size is reported and then cardinality inference and symbolic execution is repeated.

```
[Search]
partial model P2 of Digraph { }


[Search(YourOwnStrategy)]
partial model P2 of Digraph { }
```

Users can defined their own search preferences very easily.

Questions?

http://research.microsoft.com/formula

FORMULA

Modeling Foundations.