# Towards a Linear Algebra of Programming

## ( Introduction )

J.N. Oliveira

HASLAB — INESC TEC and Univ. Minho, Portugal

Thematic Seminar II
**MAPi PhD Programme 2011/12**
18–22 of January, 2012
FEUP, Porto

# Context

A Doctoral Programme on Computer Science (Informatics) brings about

- Informatics engineering
- Software engineering
- Software equality
- Software (un)predictability
- Software testing techniques
- Software modeling
- ...

# Software Engineering

Does the word **“engineering”** in phrase

**“software engineering”**

mean the same as in phrases

**“civil engineering” , “mechanic engineering”**

and so on?

(My) answer:

*Thus far (I am sorry to say. . . )* — **no**.

Why?

# What is wrong?

As Parnas (2010) writes,

> *(...) there is a disturbing gap between software development and traditional engineering disciplines.*

In such disciplines one finds a well-established maths background taught regularly at every higher-education institute, essentially made of **calculus**, **linear algebra** and **probability theory**.

Worse than this (Parnas again):

> *We must learn to use mathematics in software development, but we need to (...) be prepared to discard, most of the methods that we have been discussing and promoting for all these years.*

# Engineering mathematics

Central to engineering mathematics is the construction of sets of simultaneous **equations** as **models** of physical systems (eg. circuits, power grids),

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + a_{1m}x_m &= b_1 \\ \vdots & \vdots \quad \vdots \\ a_{n1}x_1 + a_{n2}x_2 + a_{nm}x_m &= b_n \end{cases} \tag{1}$$

that is, formulæ of the form

$$\forall\ i\ :\ 1 \leq i \leq n : \sum_{j=1}^{m} a_{ij}x_j = b_i \tag{2}$$

# Maturity

Maturity of traditional engineering mathematics:

- Engineers not intimidated by very large sets of equations.
- Thanks to the **matrix** and **vector** concepts, grouping all coefficients $a_{ij}$ of (1) in a matrix $A$, variables $x_j$ in a vector $X$ and values $b_i$ in a vector $B$, (1) becomes

$$A \cdot X \quad = \quad B$$

where operator $(\cdot)$ denotes matrix multiplication.

Backhouse (2004) writes:

> *"In this way a set of equations has been reduced to a single equation. This is a tremendous improvement in concision that does not incur any loss of precision!"*

# Quoting our "founding fathers"

Phrase **software engineering** seems to date from the Garmisch NATO conference in 1968:

> *In late 1967 the Study Group recommended the holding of a working conference on Software Engineering. The phrase 'software engineering' was deliberately chosen as being* **provocative**, *in implying the need for software manufacture to be based on the types of* **theoretical foundations** *and practical disciplines, that are traditional in the established branches of engineering.*

Question:

- Provocative or not, how "scientific" do such foundations turn out to be, 40 years later?

# Reaction

The **Garmisch NATO** conference triggered much research on how to address the so-called *software crisis*.

In the words of Brian Randell, one of the authors of the **Garmisch Report**, Edsger W. Dijkstra (1920-2002) was

> *one of a very small number of people who, through their research and teaching, have provided* **computing** *with an* **intellectual foundation** *that can justifiably be termed a* **science**.

Dijkstra's work puts emphasis on **formal logic** and **deductive reasoning** — far away from traditional engineering mathematics.

# Reaction

Much later, Bird and de Moor (1997) come up with a textbook on an **Algebra of Programming** (AoP) about which Prof. Tony Hoare (Microsoft Cambridge) writes:

> *Programming notation can be expressed by "***formulæ** *and* **equations** *(...) which share the* **elegance** *of those which underlie* **physics** *and* **chemistry** *or any other branch of basic science".*

A well-known example of such a formula is

$$Sort \subseteq Ordered \cdot Permutation$$

expressing the meaning of **sorting**.

# AoP (algebra of programming)

But, — is the meaning of the dot ($\cdot$) in

$$Sort \quad \subseteq \quad Ordered \cdot Permutation$$

the same as in linear algebra (matrix multiplication)?

In predicate logic one would write

$$l' = sort(l) \quad \Rightarrow \quad permutes(l', l) \wedge ordered(l')$$

where *sort* is some sorting algorithm and *permutes*, *ordered* are the obvious predicates.

# AoP (algebra of programming)

The intuition behind the **AoP** relies on **discrete maths** (functions, sets, relations) and **set theory**.

Emphasis on binary **relation algebra**.

Can be shown to relate to Codd's relational **database** algebra.

**AoP** is already algebraic and calculational but... not yet the **linear algebra**, **calculus** etc which stay at the foundations of the other branches of engineering.

Research question:

> *Is there a way to do logic, set theory, etc in that very same algebra which engineering as a whole is based upon?*

# Summary of seminar

In this seminar we will suggest **linear algebra** as a foundation not only of science and engineering but also of **software engineering**.

With this we hope to contribute to fulfilling the aims of the founding fathers of **software engineering** which were quoted before.

In particular, **formal logic** and **set theory** is encodable in **LA**.

However...

> Standard **LA** is unfit for such purposes and needs to be "spruced up" ;-) — more about this later.

# Down to earth, please!

**Trustworthiness** — the Holy Grail of engineering in general.

Quoting Schneider (1999)'s "Trust in Cyberspace", a trustworthy system is one that

> *(...) does what people expect it to do — and not something else — despite environmental disruption, human user and operator errors, and attacks by hostile parties.*

Furthermore,

> *Design and implementation errors must be avoided, eliminated or somehow tolerated.*

# Trustworthiness in software design

Two dual approaches to software trustworthiness:

1. **"Angelic"** — prevent bad things from happening — **weakest pre-conditions** (Dijkstra): the least one should impose for a program not blow up.

2. **"Demonic"** — force bad things to happen — **strongest post-conditions**: evaluate worst blow-up scenario arising from fault.

Example of fault injection: weaken (the weakest!!) pre-conditions.

# SWIFI

Fault-injection (Wikipedia):

> *In software testing, fault injection is a technique for improving the coverage of a test by introducing faults to test code paths, in particular error handling code paths, that might otherwise rarely be followed.*

Used used primarily as a test of the dependability on kernel software services.

Example: **SWIFI** (software injected fault-injection) code mutation such as eg. `a := a - 1` where `a := a + 1` had been written before.

# Evaluate? Quantify?

What does word **evaluate** mean in "evaluate worst blow-up scenario" above?

In practice there is no prediction at all — software tools monitor faulty software runs and gather data which, once mined, gives an evaluation.

Example (dear to the national industry):

- **Xception** by CriticalSoftware SA — SWIFI tool used for black box and white box testing
- **Xtract** — **Xception** Analysis Tool.

# Evaluate? Quantify?

Research question:

> Can worst scenarios be evaluated (quantified) **without** running the code?

No magic: this could only be done by **reasoning** about the faulty code and **calculating** (quantifying) the extent of the fault's impact.

However — reasoning about code being carried out in **logic**, how does one "quantify in logic"?

# Example: fault-injected multiplication

Safe multiplication (over $\mathbb{N}_0$): $(a*) = \textbf{for}\ (a+)\ 0$, that is,

$$a * 0 = 0$$
$$a * (n + 1) = a + a * n$$

Bad multiplication, fault-injected — 5% probability of a wrong base case (in extended functional notation):

$$a * 0 =_{.95} 0$$
$$a * 0 =_{.05} a$$
$$a * (n + 1) =_{1} a + a * n$$

Question: does the fault in the base case carry over to the overall function? In what extent? (**Quantify** fault propagation.)

# Quantitative computer science

Trend towards **quantitative methods** in computer science (using **LA** in particular):

- Read Baroni and Zamparelli (2010) suggestive paper: *Nouns are vectors, adjectives are matrices* in semantics of natural languages.
- "Quantum inspiration" in Sernadas et al. (2008) who regard probabilistic programs as linear transformations over suitable vector spaces.

Our own trend: MAPi PhD thesis by Macedo (2012) entitled

*"Matrices as Arrows" — Why Categories of Matrices Matter"*

"Arrows"? What's this? What for?

# The function-relation-matrix hierarchy

- **Functions** — rule of correspondence between inputs and outputs, eg.

$$y = f(x)$$
$$y = ax + b$$
$$y = \textit{height of } x$$

- **Relations** — multi-way, non-deterministic correspondences, eg.

$$y \textit{ likes } x$$
$$y \leq x$$

- **Matrices** — quantified relations, cf.

$$y \; M \; x = k$$

further to

$$y \; M \; x = \textit{true}$$

eg. $\textit{John likes Mary} = 99\%$ ("very much"!)

# Arrow notation for functions

Used everywhere for declaring **functions**, eg.

$$\begin{array}{lcl} f & : & \mathbb{N} \to \mathbb{R} \\ n & \mapsto & \frac{n}{\pi} \end{array}$$

The first line is the **type** of the function (**syntax**) and the second line is the rule of correspondence (**semantics**).

**Compositionality** — functions compose with each other:

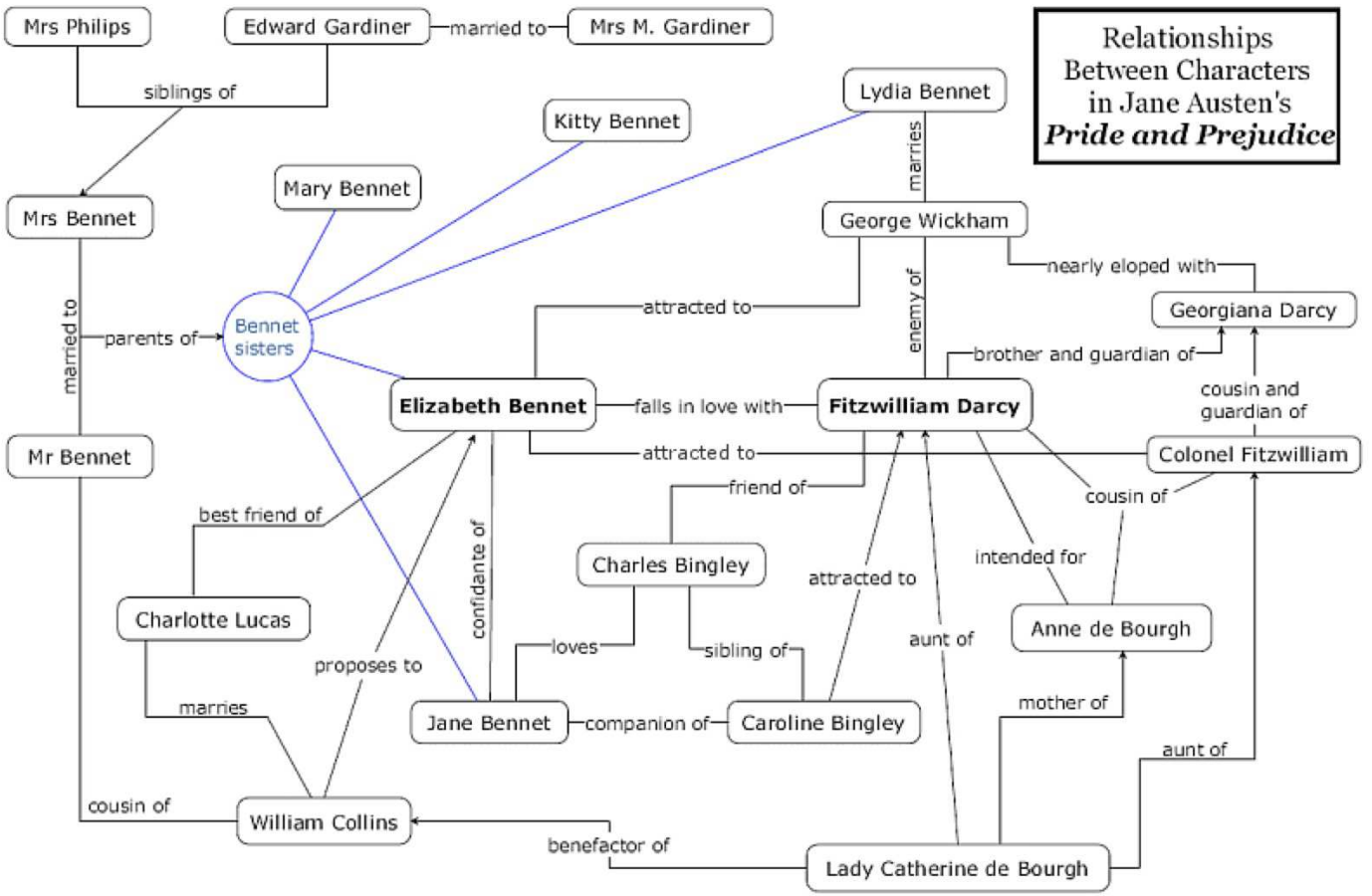
$$B \xleftarrow{f} A \xleftarrow{g} C \qquad (3)$$

$$B \xleftarrow[f \cdot g]{} C$$

$$b = f\,(g\ c)$$

# Relations

In real life, "everything is a relation" — look how book **Pride and Prejudice** (Jane Austin, 1813) is captured at Wikipedia:



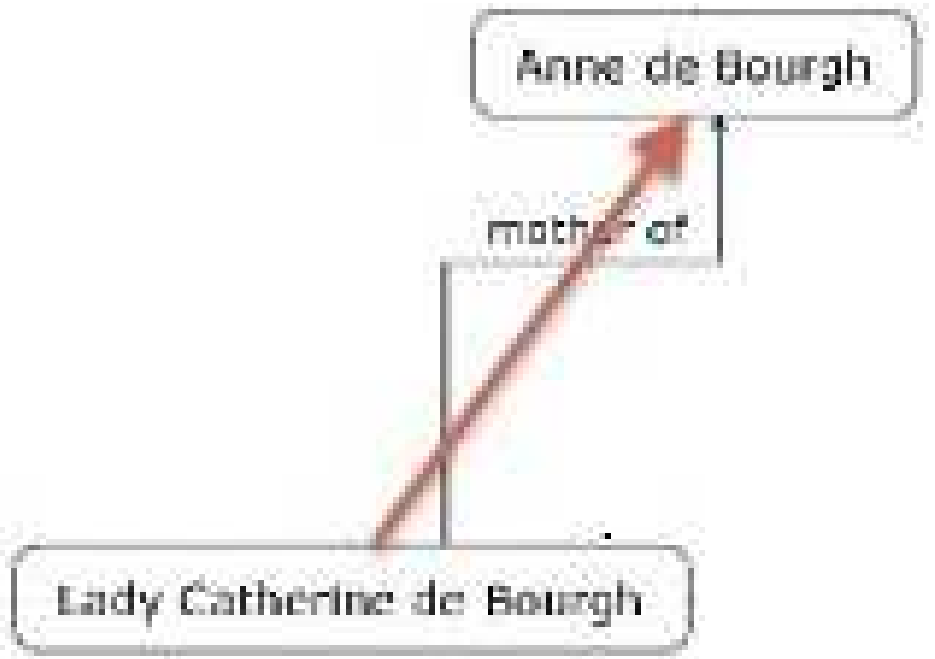
# Arrow notation for relations

The picture is a collection of relations — vulg. a **semantic network** — elsewhere known as a (binary) **relational system**.

Besides the use of **arrows** in the picture (aside) not many people would write

$mother\_of : People \rightarrow People$

as the **type** of relation *mother_of*.



# Arrow notation for (binary) relations

Binary relations are typed:

## Arrow notation

Arrow $A \xrightarrow{R} B$ denotes a binary relation from $A$ (source) to $B$ (target).

$A, B$ are types. Writing $B \xleftarrow{R} A$ means the same as $A \xrightarrow{R} B$ .

**Compositionality** — relations compose with each other:

$$B \xleftarrow{R} A \xleftarrow{S} C \qquad R \cdot S \tag{4}$$

$$b(R \cdot S)c \;\Leftrightarrow\; \langle \exists\ a \;::\; b\ R\ a \wedge a\ S\ c \rangle \tag{5}$$

Example: $Uncle = Brother \cdot Parent$

# Relations as arrows

In fact:

$$u \; Uncle \; c \;\Leftrightarrow\; \langle \exists \; p \;::\; u \; Brother \; p \wedge p \; Parent \; c \rangle$$

Question:

> *Can we make relational notation useful for specifying a reasoning about software?*

The remainder of these slides try to provide you with an affirmative answer.

# Is thinking an art?

Extract from Propositiones ad acuendos iuuenes ("Problems to sharpen the young") compiled by abbot Alcuin of York († 804):

> XVIII. Propositio de homine et capra et lvpo.
> *Homo quidam debebat ultra fluuium transferre lupum, capram, et fasciculum cauli. Et non potuit aliam nauem inuenire, nisi quae duos tantum ex ipsis ferre ualebat. Praeceptum itaque ei fuerat, ut omnia haec ultra illaesa omnino transferret. Dicat, qui potest, quomodo eis illaesis transire potuit?*

# Is thinking an art?

XVIII. Fox, goose and bag of beans puzzle. *A farmer goes to market and purchases a fox, a goose, and a bag of beans. On his way home, the farmer comes to a river bank and hires a boat. But in crossing the river by boat, the farmer could carry only himself and a single one of his purchases - the fox, the goose, or the bag of the beans. (If left alone, the fox would eat the goose, and the goose would eat the beans.) Can the farmer carry himself and his purchases to the far bank of the river, leaving each purchase intact?*

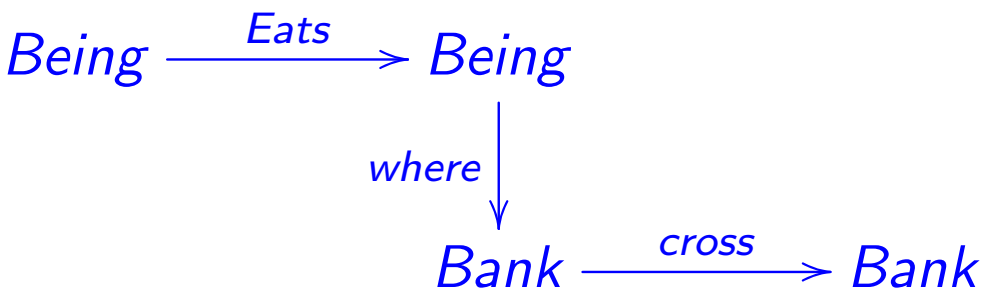
Let us identify the main **types** and **relations** involved in the puzzle and draw them in a diagram.

# Propositio de homine et capra et lvpo

Humans' mind main ability to solve problems has to do with **abstraction** — the ability to tell apart the **things** which belong to the solution from those which don't matter. For instance,

$$\begin{aligned} \mathit{Being} &= \{\mathit{Fox}, \mathit{Goose}, \mathit{Beans}, \mathit{Farmer}\} \\ \mathit{Bank} &= \{\mathit{Left}, \mathit{Right}\} \end{aligned}$$

matter, as do the relationships among them:

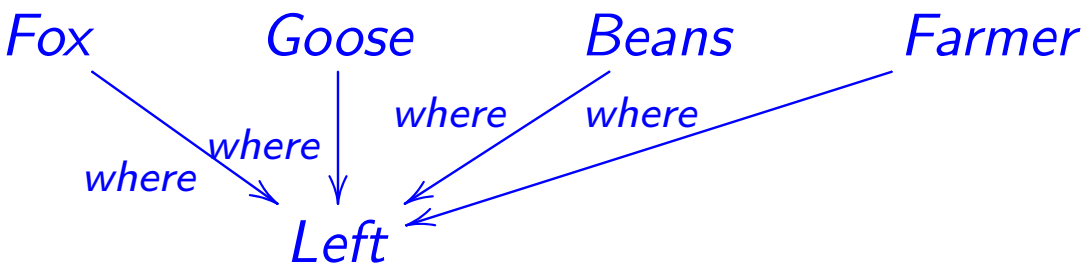


# Propositio de homine et capra et lvpo

It's easy to see that *cross* is a function which we would write eg.

$$\begin{aligned} \mathit{cross}\ \mathit{Left} &= \mathit{Right} \\ \mathit{cross}\ \mathit{Right} &= \mathit{Left} \end{aligned}$$

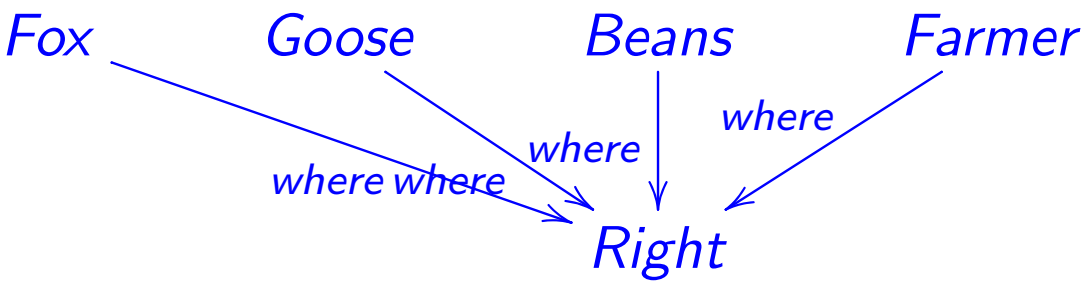
in a functional programming language. In the beginning, all beings are on the same bank, eg. *Left*:




(initial state of the relational model)

# Propositio de homine et capra et lvpo

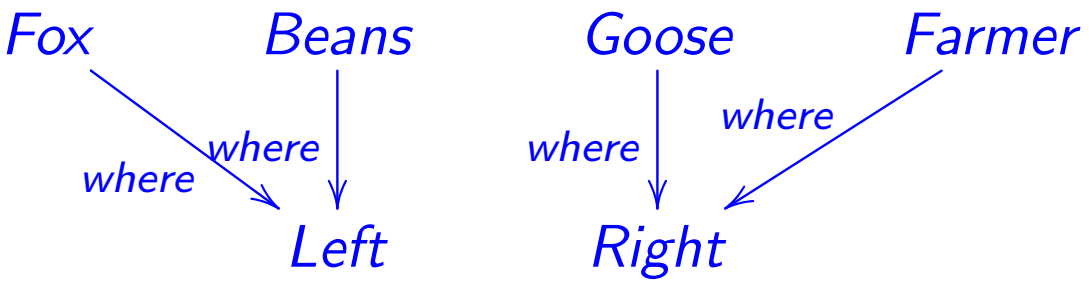
At the end (should there be an end...) we want them all on the other bank:



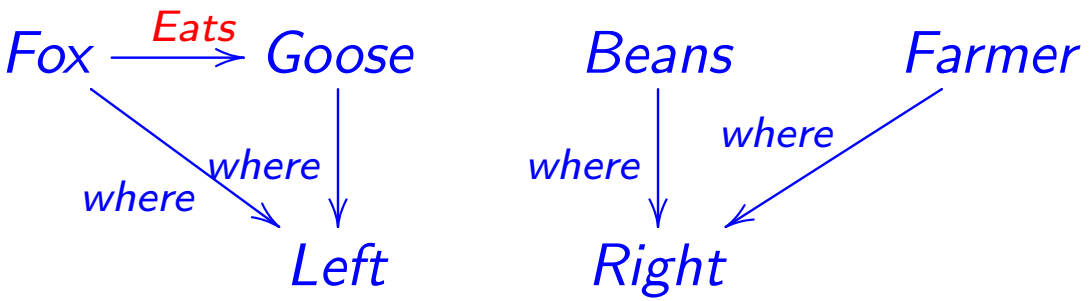

(final state of the relational model).

# Propositio de homine et capra et lvpo

In between there are **safe** states (=**legal**), for instance




but there is always the risk of moving to an unsafe state (=**illegal**) as, for instance,



# Propositio de homine et capra et lvpo

Relation *Eats* is the obvious food chain

$$Fox > Goose > Beans$$

Observations concerning *where*: one will always say, eg.

*"the" bank where the Goose is*

and not

*"a" bank where where the Goose may be*

This happens because beings are **always** at one **and only one** bank.

In the same way we say that, when crossing the river,

*one crosses to "the" other bank*

and not to "another" bank.

# Propositio de homine et capra et lvpo

Likewise, we say

> 6 *is ("the" outcome of) "the" product of* 2 *by* 3
> $(6 = 2 * 3)$

and not

> 6 *is "a" (possible) outcome of multiplying* 2 *by* 3

This is so because, like *cross* and *where*, the multiplication of two numbers

- always exists (**existence**)
- is one **and the same** number (**uniqueness**).

# Propositio de homine et capra et lvpo

**Existence** and **uniqueness** make the usual notation

$$y = f(x)$$

meaningful, making it mathematically meaningful to substitute $y$ for $f(x)$ wherever it occurs. From the linguistic perspective,

> *functions in mathematics and modeling are related to the use of* **definite articles** *in natural languages.*

As counter-example take, for instance, one saying that $2$ is "a" square root of $4$, for there is another one: $-2$. This, notation $2 = \sqrt{4}$ often found in textbooks is incorrect.

# Functions are relations

- We regard function $f : A \to B$ as the binary relation which relates $b$ to $a$ iff $b = f\ a$. So, $b\ f\ a$ literally means $b = f\ a$.
- Therefore, we specialize

$$B \xleftarrow{R} A \qquad b\ R\ a$$

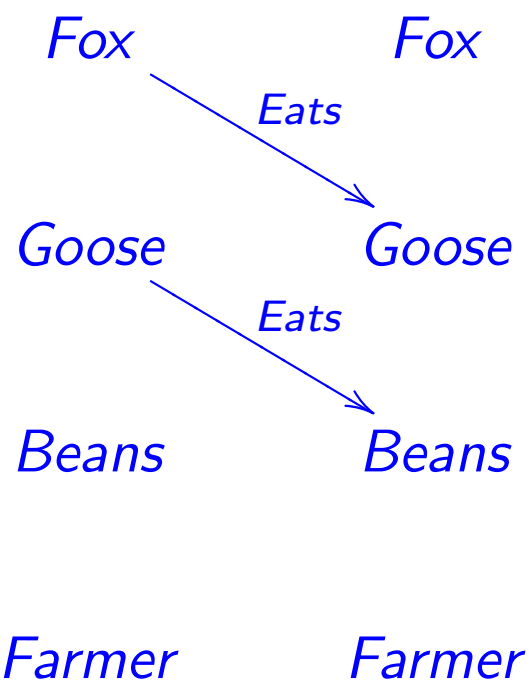
to

$$B \xleftarrow{f} A \qquad b = f\ a$$

- Lowercase letters (or identifiers starting by one such letter) will denote **functions**, eg. $f$, $g$, $succ$, etc.

# Partial functions

Let us now inspect relation *Eats*:




(as *Farmer*'s eating habits are irrelevant to the problem).

**Question:** *is Eats a function (over Being)?*

# Partial functions

One can observe that **uniqueness** holds (frugality: any $x$ eats at most one $y$) but **existence** doesn't: $g$ don't eat anything, for instance.

- Relations of this kind are known as **partial** functions or **simple** relations
- They are ubiquitous in maths and computing.
- They can be regarded as **deterministic** relations or as "functions" which are undefined for some of its inputs.
- As data structures, **simple** relations establish **primary key** relationships, eg. $Individual \overset{Passport}{\longleftarrow} \mathbb{N}$ — not every passport number is in use + no two passports have the same number.
- Functional dependencies in databases are **simple** relations.

# Propositio de homine et capra et lvpo

Naturally, we may build new relations out of existing ones, for instance:

$$Being \xrightarrow{SameBank} Being$$

which is easy to define:

$$b\ SameBank\ a \quad \textbf{if and only if} \quad where(b) = where(a)$$

Another example,

$$Being \xrightarrow{CanEat} Being$$

defined by:

$$b\ CanEat\ a \quad \textbf{if and only if} \quad (b\ SameBank\ a)\ \textbf{and}\ (b\ Eats\ a)$$

# Propositio de homine et capra et lvpo

The usual symbol in relation algebra for denoting this situation is:

$$CanEat \quad = \quad SameBank \ \cap \ Eats$$

In general, given two relations $R$ and $S$, define

$$b(R \cap S)a \quad \Leftrightarrow \quad b \ R \ a \ \wedge \ b \ S \ a$$

This relational combinator $R \cap S$ is known as **intersection** and it captures two **simultaneous** relationships among the same objects (one $R$ and the other $S$).

# Propositio de homine et capra et lvpo

The most important ingredient of the problem is the property which, in words, reads as follows:

> *If anybody can eat somebody* **then** *the farmer should be on that bank (in presence of the farmer animals are forced to starving)*

Properties such as this, which we want to hold at any time, whatever moves are made, are known as **invariant** properties (ie. the model may change state but only within the validity space of its invariants).

It's easy to express the above **invariant** property using already defined relationships:

$$\mathit{CanEat} \quad \textbf{only if} \quad \mathit{SameBank} \cdot \underline{\mathit{Farmer}}$$

(Notation $\underline{\mathit{Farmer}}$ is explained below.)

# Propositio de homine et capra et lvpo

First note the expression

$$R \quad \textbf{only if} \quad S$$

whose meaning is

*for all $b, a$, wherever $b\ R\ a$ holds, then $b\ S\ a$ also holds.*

Usual notation and definition:

$$R \subseteq S \quad \Leftrightarrow \quad \langle \forall\ b, a\ :\ b\ R\ a :\ b\ S\ a \rangle \tag{6}$$

(read $R \subseteq S$ as "$R$ is at most $S$")

Notation *Farmer* is an instance of a **constant** function. In general, given a non-empty set $K$ and some $k \in K$, we have

$$y\ \underline{k}\ x \quad \Leftrightarrow \quad y = k$$

# Propositio de homine et capra et lvpo

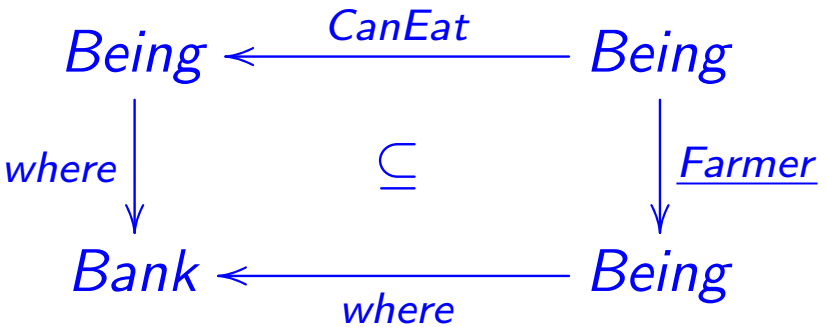
Later we will learn to prove that the given invariant is the same as

$$\mathit{where} \cdot \mathit{CanEat} \quad \subseteq \quad \mathit{where} \cdot \underline{\mathit{Farmer}}$$

In words:

> **Where** *one can eat (somebody) that's* **where** *the farmer is*

— which can be drawn in the following way, thanks to the arrow view of relations:




depicting the relationships involved.

# Summing up

Questions:

- How advantageous are the expressions and diagrams above in solving the puzzle?
- Is there a **programming language** helping us to find a solution?

Answer to the first question:

- The notation given so far is that of so-called **Relational Mathematics**, which enables a calculational style similar to that one is used in solving systems of equations in algebra.

# Relational Mathematics

Relational maths finds its roots in the pioneering work

> *On the syllogism: IV, and on the logic of relations*

read by the British mathematician Augustus de Morgan (1806-71), on the 23rd April 1860 to the Cambridge Philosophical Society.
(Excerpts of this work follow.)

# Augustus de Morgan (1806-71)

Binary relations:

> [...] *Let $X..LY$ signify that $X$ is some one of the objects of thought which stand to $Y$ in the* **relation** *$L$, or is one of the $L$s of $Y$.*

Relational composition:

> [...] *When the predicate is itself the subject of a relation, there may be a* **composition***: thus if $X..L(MY)$, if $X$ be one of the $L$s of one of the $M$ s of $Y$, we may think of $X$ as an '$L$ of $M$' of $Y$, expressed by $X..(LM)Y$, or simply by $X..LMY$.* [...][So] *brother of parent is identical with uncle, by mere definition.*

Relational converse:

> [...] *The* **converse** *relation of $L$, $L^{-1}$, is defined as usual: if $X..\ L\ Y$, $Y\ ..\ L^{-1}\ X$ : if $X$ be one of the $L$s of $Y$, $Y$ is one of the $L^{-1}$ s of $X$.*

# Bad fate

As Maddux (1991); Givant (2006) explain:

- Charles Peirce (1839-1914) invented **quantifier** notation to explain de Morgan's algebra of relations.
- Further (monumental) contribution by Ernst Schröder (1841-1902) eventually led to first order **logic** (FOL) itself.

However, and in spite of Bertrand Russell (1872-1970)'s writing

> [...] *The subject of symbolic logic is formed by three parts: the calculus of propositions, the calculus of classes, and the* **calculus of relations**

in his *Principles of Mathematics* (1903), the language (FOL) invented to explain the calculus of relations became eventually more popular than the calculus itself.

# Formal specification (modeling) languages

Answer to the second question:

- The languages which help at this high level of abstraction are no longer conventional **programming** languages, but rather those known as **formal specification** languages.
- In such languages we tell the machine **what** we want it to achieve. rather that prescribing the details (machine instructions) on **how** to do it.
- The equivalent to **interpreters** at this level are tools known as **model checkers**.
- Instead of computing and printing results, a **model checker** helps us to see whether **what** we intend to achieve makes sense (cf. contradictions, ambiguities etc).

# Alloy

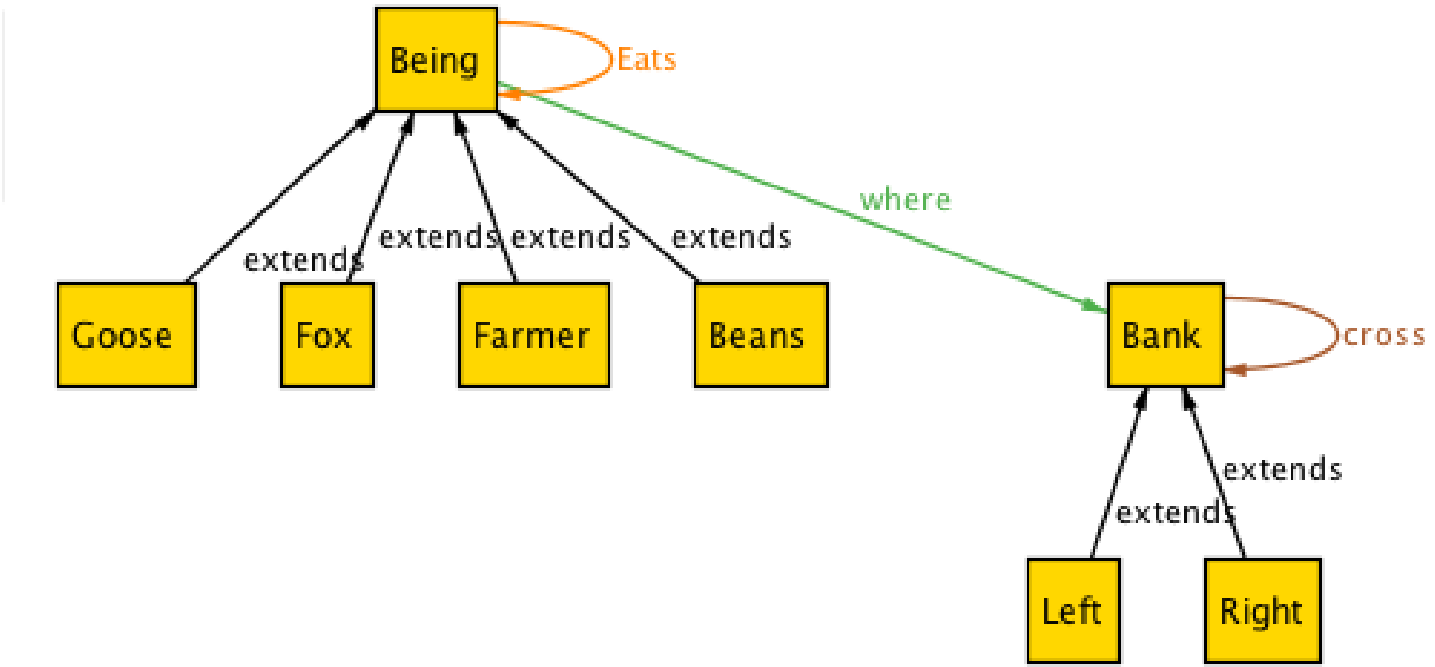
**Alloy Analyser** is a **model checker** which uses relational algebra as core notation. It has been developed at M.I.T. (Boston, Mass.) by a group lead by Daniel Jackson (1963-).

It therefore is a model checker specially devoted to Relational Mathematics, as captured by its lemma

> *"(...) in Alloy*
> *everything is a relation"*

# Propositio de homine et capra et lvpo

Here is how the model of our puzzle above is perceived by the **Alloy Analyser**:




The arrows A extends B should be read as: *A is a B*.

# Propositio de homine et capra et lvpo

Here is how diagram

$$Being \xrightarrow{Eats} Being \xrightarrow{where} Bank \xrightarrow{cross} Bank$$

is captured in **Alloy** notation,

```
abstract sig Being { Eats : set Being }

abstract sig Bank { cross: Bank }
```

and how we declare the particular beings and banks:

```
one sig Farmer, Beans, Goose, Fox extends Being {}
one sig Left, Right extends Bank {}
```

Later we will see how to declare relation *where*.

# Propositio de homine et capra et lvpo

Food chain

$$Fox \xrightarrow{Eats} Goose \xrightarrow{Eats} Beans$$

is written explicitly in **Alloy** as a **fact**:

```
fact { Eats = Fox -> Goose + Goose -> Beans }
```

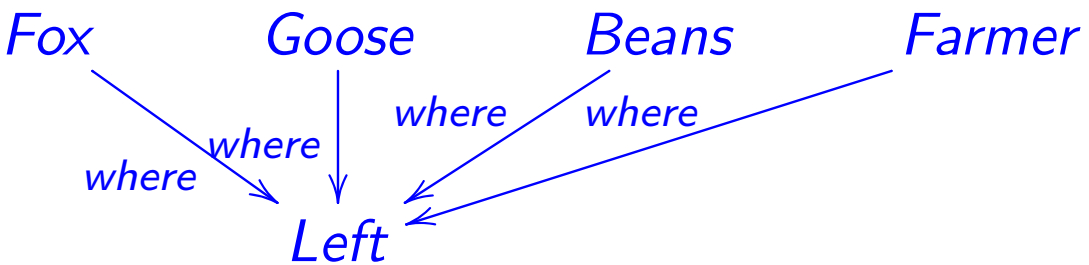
Likewise, *cross* is another fact in Alloy:

```
fact { cross = Left -> Right  +
               Right  -> Left
     }
```

# Propositio de homine et capra et lvpo

We proceed to showing how to model the **dynamic** part of the problem, that is, how to specify the **steps** which need be carried out to model check the evolution of the puzzle.

Note how, in each step ("move") the only entity which changes is function *where*, beginning at




and stopping when *where* is the other possible constant function of its type (everybody at the *Right* bank).

# Propositio de homine et capra et lvpo

The **moves** of the game are transitions among steps, that is, **states** of an automaton, In this case,

```
sig State { where : Being -> one Bank }
```

where qualifier *one* is Alloy's way of telling that *where* is a function.

What kind of "move" can we do? One, in fact: letting the *Farmer* choose one *Being* from his bank and take it to the other bank:

$before \xrightarrow{trip} after$ *is the move:*

**let** *a* **such that** *a SameBank Farmer* (*before*),
*whoMoves* = *Farmer* + *a* (*before*)
**in** *transit*[*before*, *after*, *cross*, *whoMoves*]

# Propositio de homine et capra et lvpo

In **Alloy** notation we write this in the following way:

```
pred trip[s,s' : State] {
    some c: (s.SameBank).Farmer |
            let fc = (Farmer + c) <: s.where |
                s'.where = s.where ++ fc.cross
}
```

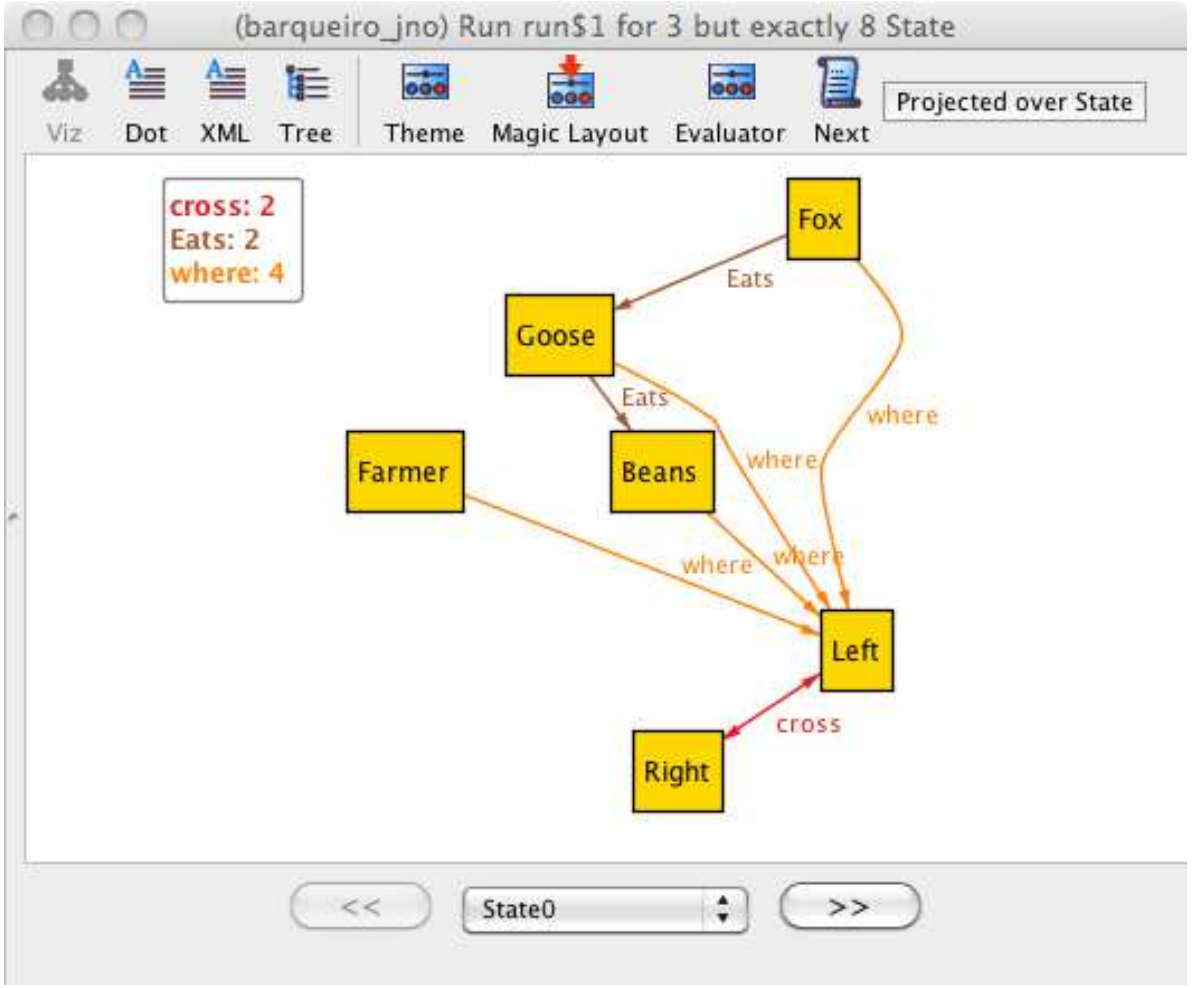
We won't go into notation details at this point, sufficing to see how the rules of the game are written in **Alloy** syntax:

```
fact {
  all s : State | starving[s]
  first.where = Being->Left
  last.where = Being->Right
  all s : State, s' : s.next | trip[s,s']
}
```

where *starving* is the **invariant** drawn as a diagram earlier on.

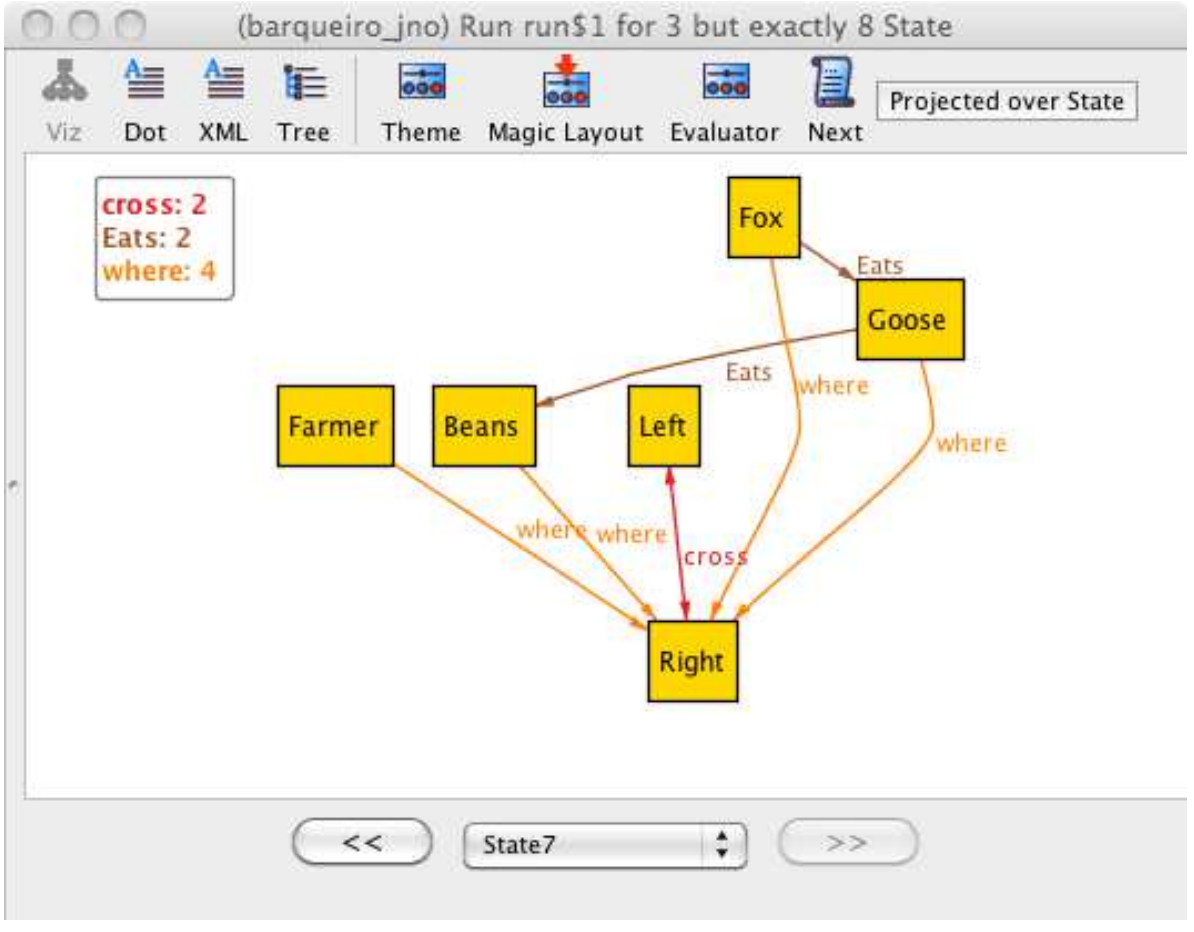
# Propositio de homine et capra et lvpo

Demo (start state):



# Propositio de homine et capra et lvpo

Demo (final state):



# Summary and prospects for what next

We have seen how **relational mathematics** naturally follows natural language in problem modeling.

We have also seen a notation — **Alloy** — which captures such models.

We have also seen a tool — **Alloy Analyser** which model-cheks such models.

However:

- Model-checking is incomplete **verification** — it can only show the **presence** of errors in modeling, not their absence.
- **Relational algebra** (vulg. **AoP**, algebra of programming) will compensate for this, see the workflow of the next slides.

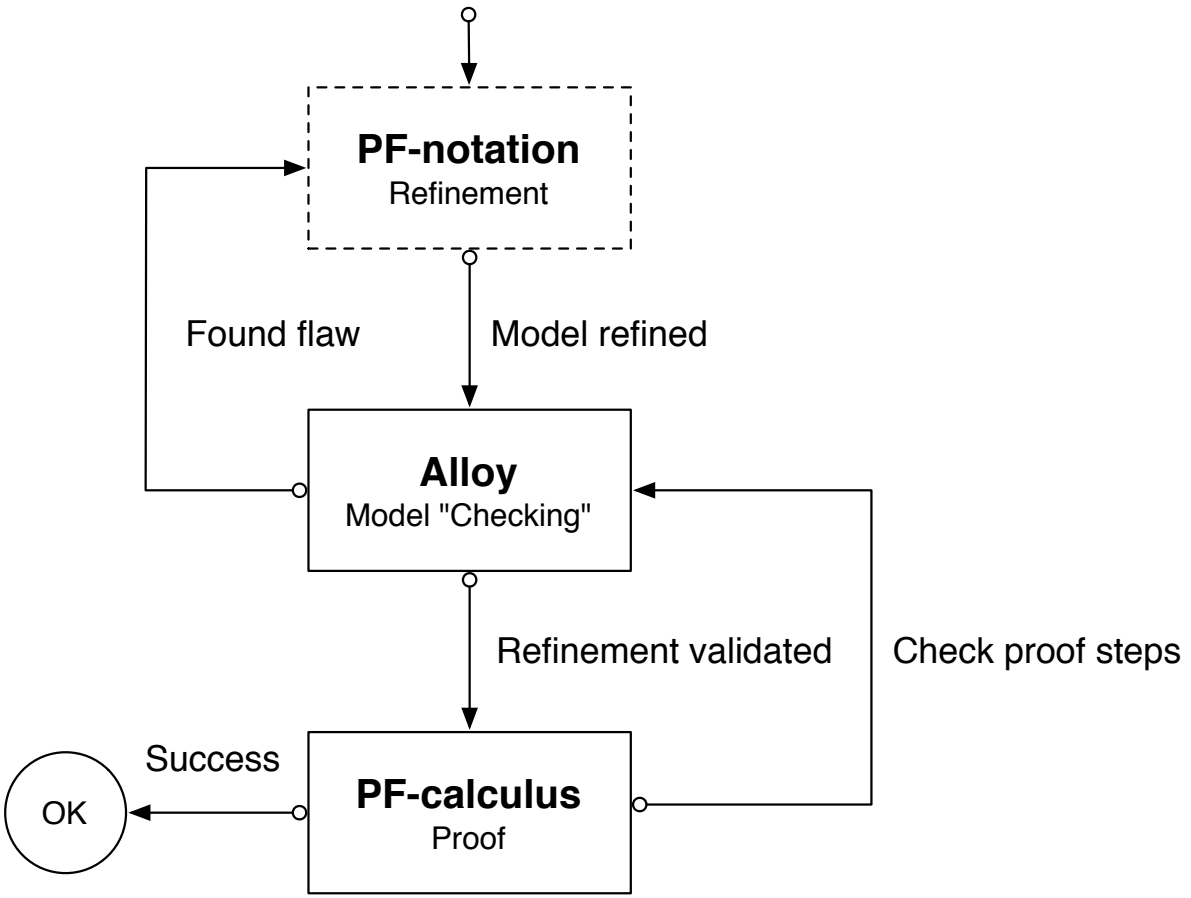
# What next — Alloy meets the AoP

The "Alloy-meets-AoP" approach to software design advocated in this seminar puts together two trends in software validation which usually do not interact with each other:


(7)

# What next — Alloy meets the AoP

Sketch of a life-cycle:



# What next — From the AoP to the LAoP

Further on:

- **relations** are Boolean **matrices**;
- when generalizing to arbritrary matrices we move from **qualitative** modeling to **quantitative** models ...
- ... towards a *Linear Algebra of Programming* (**LAoP**).

# References

R.C. Backhouse. *Mathematics of Program Construction*. Univ. of Nottingham, 2004. Draft of book in preparation. 608 pages.

M. Baroni and R. Zamparelli. Nouns are vectors, adjectives are matrices: representing adjective-noun constructions in semantic space. In *Proceedings*, EMNLP '10, pages 1183–1193, Morristown, NJ, USA, 2010. Association for Computational Linguistics.

R. Bird and O. de Moor. *Algebra of Programming*. Series in Computer Science. Prentice-Hall International, 1997.

S. Givant. The calculus of relations as a foundation for mathematics. *J. Autom. Reasoning*, 37(4):277–322, 2006. ISSN 0168-7433. doi: http://dx.doi.org/10.1007/s10817-006-9062-x.

H. Macedo. *Matrices as Arrows — Why Categories of Matrices Matter*. PhD thesis, University of Minho, 2012. (Submitted Jan. 2012).

R.D. Maddux. The origin of relation algebras in the development and axiomatization of the calculus of relations. *Studia Logica*, 50:421–455, 1991.

David Lorge Parnas. Really rethinking "formal methods". *IEEE Computer*, 43(1):28–34, 2010.

Fred B. Schneider. *Trust in Cyberspace*. The National Academies Press, 1999. ISBN 9780309131827. URL `http://www.nap.edu/openbook.php?record_id=6161`. Committee on Information Systems Trustworthiness, Commission on Physical Sciences, Mathematics, and Applications, National Research Council.

A. Sernadas, J. Ramos, and P. Mateus. Linear algebra techniques for deciding the correctness of probabilistic programs with bounded resources. Technical report, SQIG - IT and IST - TU Lisbon, 1049-001 Lisboa, Portugal, 2008. Short paper presented at LPAR 2008, Doha, Qatar. November 22-27.