Pointfree Alloy: the other side of the moon

J.N. Oliveira

Dept. Informática,
Universidade do Minho
Braga, Portugal

CIC'09
May 2009
Braga, Portugal

Model driven engineering

- **MEDEA** project — *High Assurance MDE using Alloy*
- **MDE** is a clumsy area of work, full of approaches, acronyms, notations.
- **UML** has taken the lead in *unifying* such notations, but it is too **informal** to be accepted as a reference approach.
- Model-oriented formal methods (**VDM**, **Z**) solve this informality problem at a high-cost: people find it hard to understand models written in maths (cf. maths illiteracy if not mathphobic behaviour).
- **Alloy** [4] offers a good compromise — it is formal in a light-weight manner.

Inspiration

- **BBI** project [5]: **Alloy** re-engineering of a well-tested, very well written non-trivial prototype in **Haskell** of a real-estate trading system similar to the stocks market (65 pages in lhs format) unveiled 4 bugs (2 invariant violations + 2 weak pre-conditions)
- Alloy and Haskell complementary to each other

Alloy

What **Alloy** offers

- A unified approach to **modeling** based on the notion of a **relation — "everything is a relation"** in Alloy.
- A minimal syntax (centered upon relational composition) with an object-oriented flavour which captures much of what otherwise would demand for **UML**+**OCL**.
- A **pointfree** subset.
- A model-checker for model assertions (counter-examples within scope).

Alloy

What **Alloy does not** offer

- Complete calculus for deduction (proof theory)
- Strong type checking
- Dynamic semantics modeling features

Opportunities

- Enrich the standard Alloy *modus operandi* with relational algebra calculational proofs
- Connect the tool to a theorem prover, eg. **Prover9** as suggested in [3]
- Design an Alloy-centric tool-chain for high assurance model-oriented design

Thus the **MEDEA** project (submitted).

Relational composition

- The swiss army knife of Alloy
- It subsumes function application and “field selection”
- Encourages a navigational (point-free) style based on pattern $x.(R.S)$.
- Example:

$$
\begin{aligned}
Person &= \{(P1),(P2),(P3),(P4)\} \\
parent &= \{(P1,P2),(P1,P3),(P2,P4)\} \\
me &= \{(P1)\} \\
me.parent &= \{(P2),(P3)\} \\
me.parent.parent &= \{(P4)\} \\
Person.parent &= \{(P2),(P3),(P4)\}
\end{aligned}
$$

When "everything is a relation"

- Sets are relations of arity 1, eg.
  $Person = \{(P1), (P2), (P3), (P4)\}$
- Scalars are relations with size 1, eg. $me = \{(P1)\}$
- Relations are first order, but there are multi-ary relations.
- However, **Alloy** relations are not *n*-ary in the usual sense: instead of thinking of $R \in 2^{A \times B \times C}$ as a set of triples (there is no such thing as *tupling* in Alloy), think of R in terms of *currying*:

  $$R \in (B \to C)^A$$

  (More about this later.)

Kleene algebra flavour

Basic operators:

.	*composition*
+	*union*
^	*transitive closure*
*	*transitive-reflexive closure*

(There is no explicit recursion is **Alloy**.) Other relational operators:

~	*converse*
++	*override*
&	*intersection*
-	*difference*
->	*cartesian product*
<:	*domain restriction*
:>	*range restriction*

Relational thinking

- As a rule, thinking in terms of poinfree relations (this includes **functions**, of course) pays the effort: the concepts and the reasoning become simpler.
- This includes **relational data** structuring, which is far more interesting than what can be found in SQL and relational databases.

Example — list processing

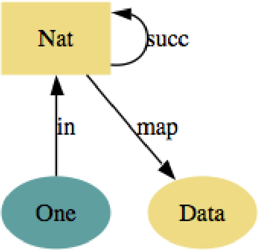
- **Lists** are traditionally viewed as recursive (linear) data structures.
- There are no lists in Alloy — they have to be modeled by **simple** relations (vulg. partial functions) between indices and elements.

Lists as relations in Alloy

```
sig List {
        map : Nat -> lone Data
}

sig Nat {
    succ: one Nat
}

one sig One in Nat {}
```



Multiplicities: `lone` (one or less), `one` (exaclty one)

Relational data structuring

Some correspondences:

list l	**relation** L
sorted	monotonic
no-duplicates	injective
$map\ f\ l$	$f \cdot L$
$zip\ l_1\ l_2$	$\langle L_1, L_2 \rangle$
$[1, \ldots]$	id

where

- id is the identity (equivalence) relation (also a function)
- the "fork" (also known as "split") combinator $\langle \_, \_ \rangle$ is such that $(x, y)\langle L_1, L_2 \rangle z$ means the same as $x L_1 z \wedge y L_2 z$

Haskell versus Alloy

Pointwise Haskell:

```
findIndices       :: (a -> Bool) -> [a] -> [Int]
findIndices p xs = [ i | (x,i) <- zip xs [0..], p x ]
```

Pointfree (PF):

$$findIndices\ p\ L \quad \triangleq \quad \pi_2 \cdot (\Phi_p \times id) \cdot \langle L, id \rangle \qquad (1)$$

where

- π_2 is the right projection of a pair
- $L \times R = \langle L \cdot \pi_1, R \cdot \pi_2 \rangle$
- $\Phi_p \subseteq id$ is the coreflexive relation (partial identity) which models predicate p (or a set)

Haskell versus Alloy

- What about Alloy? It has no pairs, therefore no forks $\langle L, R \rangle$...
- Don't worry — Alloy is relational and we can play with the relational calculus:

$$\begin{array}{cl} & \pi_2 \cdot (\Phi_p \times id) \cdot \langle L, id \rangle \\ = & \{ \ \times\text{-absorption} \ \} \\ & \pi_2 \cdot \langle \Phi_p \cdot L, id \rangle \\ = & \{ \ \times\text{-cancelation} \ \} \\ & \delta\,(\Phi_p \cdot L) \end{array}$$

where δ is the **domain** operator: $\delta\, R = R^{\circ} \cdot R \cap id$, for R° the converse of R.

Haskell versus Alloy

Two ways of writing $\delta\,(\Phi_p \cdot L)$ in Alloy, one pointwise

```
fun findIndices[s:set Data,l:List]: set Nat {
      {i: Nat | some x:s | x in i.(l.map)}
}
```

and the other pointfree,

```
fun findIndices[s:set Data,l:List]: set Nat {
      dom[l.map :> s]
}
```

the latter corresponding to what we've calculated.

Beyond model-checking: proofs by calculation

Suppose the following property

$$(findIndices\ p) \cdot r^{\star} \quad = \quad findIndices\ (p \cdot r) \qquad (2)$$

is asserted in (pointwise) Alloy:

```
assert FT {
    all l,l':List, p: set Data, r: Data -> one Data |
        l'.map = l.map.r =>
            findIndices[p,l'] = findIndices[r.p,l]
}
```

NB: the following version of (2) explains the encoding above:

$$r^{\star} \subseteq (findIndices\ p)^{\circ} \cdot findIndices\ (p \cdot r)$$

whereby, going pointwise, we get

$$l' = r^{\star}\ l \ \Rightarrow\ findIndices\ p\ l' = findIndices(p \cdot r)\ l$$

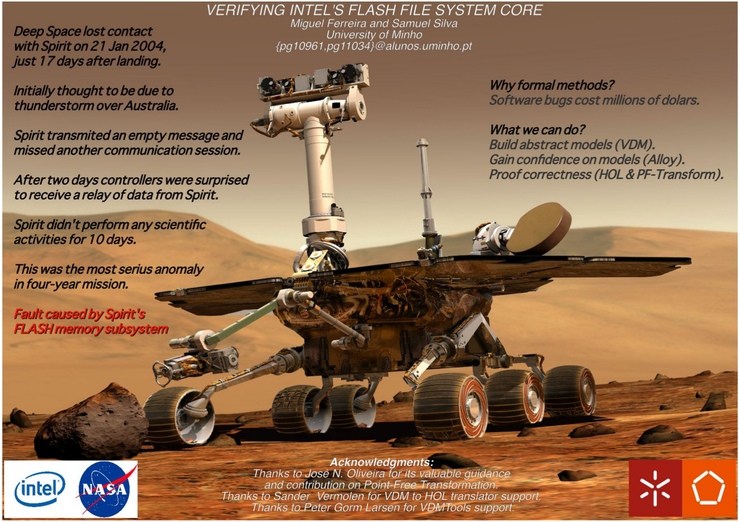
Beyond model-checking: proofs by calculation

- Suppose the **Alloy** model checker does not yield any counter-examples for this property, for increasing bounds.
- How can we be sure of its validity?
  - Free theorems — the given assertion is a corollary of the **free-theorem** [6] of *findIndices*, thus there is nothing to prove (model checking could altogether be avoided!)
  - Wishing to prove the assertion anyway, one calculates:

Trivial proof

$$
\begin{array}{ll}
 & (\mathit{findIndices}\ p) \cdot r^{\star} = \mathit{findIndices}\ (p \cdot r) \\
\Leftrightarrow & \quad \{\ \text{list to relation transform}\ \} \\
 & \delta\,(\Phi_p \cdot (r \cdot L)) = \delta\,(\Phi_{p \cdot r} \cdot L) \\
\Leftrightarrow & \quad \{\ \text{property } \Phi_{f \cdot g} = \delta\,(\Phi_f \cdot g)\ \} \\
 & \delta\,(\Phi_p \cdot (r \cdot L)) = \delta\,(\delta\,(\Phi_p \cdot r) \cdot L) \\
\Leftrightarrow & \quad \{\ \text{domain of composition}\ \} \\
 & \delta\,(\Phi_p \cdot (r \cdot L)) = \delta\,((\Phi_p \cdot r) \cdot L) \\
\Leftrightarrow & \quad \{\ \text{associativity}\ \} \\
 & \textsc{True}
\end{array}
$$

Realistic example — Verified FSystem (VFS)



VFS in Alloy (simplified)

The system:

```
sig System {
  fileStore: Path -> lone File,
  table: FileHandle -> lone OpenFileInfo
}
```

Paths:

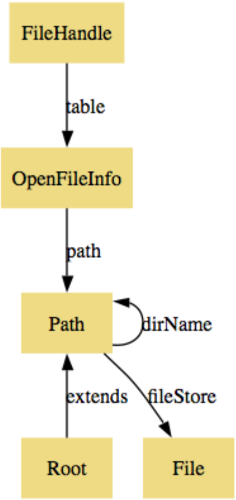
```
sig Path {
  dirName: one Path
}
```

The root is a path:

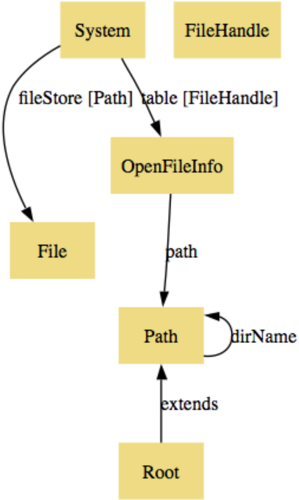
```
one sig Root extends Path {
}
```

Alloy diagrams for FSystem

Simplified:



Complete:



Binary relation semantics

Meaning of signatures:

```
sig Path {
  dirName: one Path
}
```

declares function $Path \xrightarrow{dirName} Path$.

```
sig System {
  fileStore: Path -> lone File,
}
```

declares simple relation $System \times Path \xrightarrow{fileStore} File$.
(**NB**: a relation S is **simple**, or *functional*, wherever its **image** $S \cdot S^\circ$ is coreflexive. Using harpoon arrows $\rightharpoonup$ for singling these out.)

Binary relation semantics

- Since

$$(A \times B) \rightharpoonup C \quad \cong \quad (B \rightharpoonup C)^A$$

fileStore can be alternatively regarded as a function in $(Path \rightharpoonup File)^{System}$, that is, for $s : System$,

$$Path \xrightarrow{(fileStore\ \ s)} File$$

- Thus the "navigation-styled" notation of **Alloy**: $p.(s.fileStore)$ means the file accessible from path p in file system s.
- Similarly, line `table: FileHandle -> lone OpenFileInfo` in the model declares

$$FileHandle \xrightarrow{(table\ \ s)} OpenFileInfo$$

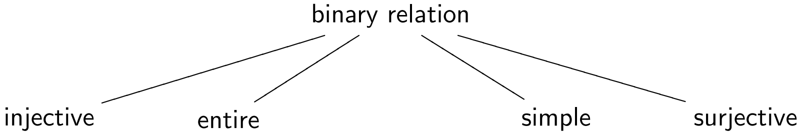
Multiplicities in Alloy + taxonomy

<table>
<tr><td colspan="3">A lone -> B</td><td colspan="3">A -> some B</td><td colspan="3">A -> lone B</td><td colspan="3">A some -> B</td></tr>
<tr><td colspan="3">injective</td><td colspan="3">entire</td><td colspan="3">simple</td><td colspan="3">surjective</td></tr>
<tr><td colspan="4">A lone -> some B</td><td colspan="4">A -> one B</td><td colspan="4">A some -> lone B</td></tr>
<tr><td colspan="4">representation</td><td colspan="4">function</td><td colspan="4">abstraction</td></tr>
<tr><td colspan="6">A lone -> one B</td><td colspan="6">A some -> one B</td></tr>
<tr><td colspan="6">injection</td><td colspan="6">surjection</td></tr>
<tr><td colspan="12">A one -> one B</td></tr>
<tr><td colspan="12">bijection</td></tr>
</table>

(courtesy of Alcino Cunha, Alloy expert at Minho)

The same — mathematically

Topmost criteria:



Definitions:

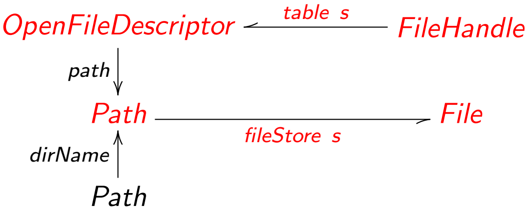
	Reflexive	*Coreflexive*
ker R	entire R	injective R
img R	surjective R	simple R

$$\ker R = R^\circ \cdot R$$

$$\operatorname{img} R = R \cdot R^\circ$$

From Alloy to relational diagrams

We draw



where

- *table s* and *fileStore s* are simple relations
- the other arrows depict functions

(Diagram in the **Rel** allegory [1] to be completed soon.)

Model constraints

Referential integrity:

Non-existing files cannot be opened:

```
pred ri[s: System]{
    all h: FileHandle, o: h.(s.table) |
                  some (o.path).(s.fileStore)
}
```

Paths closure:

Mother directories exist and are indeed directories:

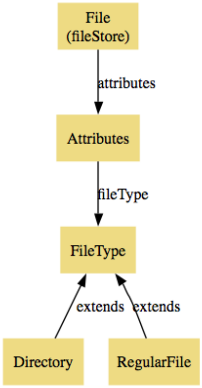
```
pred pc[s: System]{
  all p: Path |
        some p.(s.fileStore) =>
        (some d: (p.dirName).(s.fileStore) |
                          d.fileType=Directory)
}
```

2nd part of Alloy FSystem model

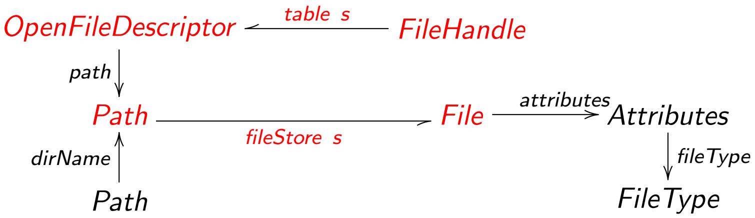
```
sig File {
    attributes: one Attributes
}

sig Attributes{
    fileType: one FileType
}

abstract sig FileType {}
one sig RegularFile extends FileType {}
one sig Directory extends FileType {}
```



Updated binary relational diagram



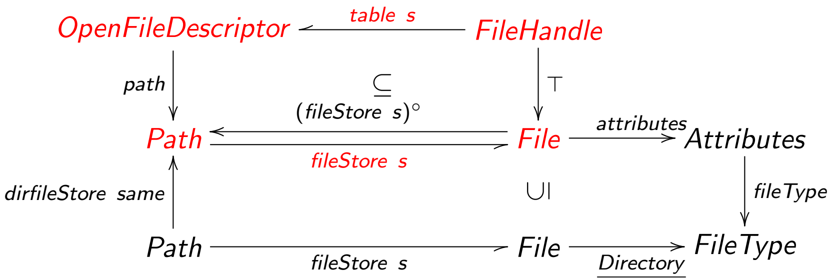
where

- *table s*, *fileStore s* are simple relations
- all the other arrows depict functions

Constraints: still missing

Updating diagram with constraints

Complete diagram, where *Directory* is the "everywhere-*Directory*" constant function:



Constraints:

- Top rectangle is the PF-transform of *ri* (referential integrity)
- Bottom rectangle is the PF-transform of *pc* (path closure)

PF-constraints in symbols

Referential integrity:

$$ri(s) \;\triangleq\; path \cdot (table\ s) \subseteq (fileStore\ s)^\circ \cdot \top \tag{3}$$

which is equivalent to

$$ri(s) \;\triangleq\; \rho\,(path \cdot (table\ s)) \subseteq \delta\,(fileStore\ s)$$

where $\rho\, R = \delta\, R^\circ$. PF version (3) nicely encoded in **Alloy**

```
pred riPF[s: System]{
    s.table.path in (FileHandle->File).~(s.fileStore)
}
```

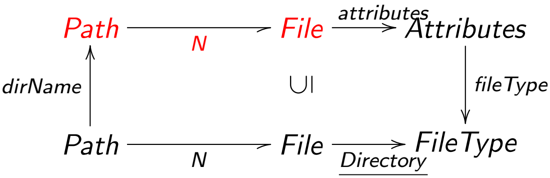
thanks to its emphasis on **composition**.

PF-constraints in symbols

Paths closure:

$$pc\ N \triangleq \underline{Directory} \cdot N \subseteq fileType \cdot attributes \cdot N \cdot dirName \qquad (4)$$

recall lower part of diagram:



Again thanks to emphasis on **composition**, this is easily encoded in PF-Alloy:

```
pred pcPF[s: System]{
   s.fileStore.(File->Directory) in
        dirName.(s.fileStore).attributes.fileType
}
```

PF-ESC by calculation

- Models with constraints put the burden on the designer to ensure that operations **type-check** (read this in **extended**-mode), that is, constraints are preserved across the models operations.
- Typical approach in MDE: **model-checking**
- Automatic **theorem proving** also considered in safety-critical systems
- However: convoluted pointwise formulæ often lead to failure.

How about doing these as "pen & paper" exercises?

- **PF-formulæ** are manageable, this is the difference.

Example of PF-ESC by calculation

Consider the operation which removes file system objects, as modeled in Alloy:

```
pred delete[s',s: System, sp: set Path]{
        s'.table = s.table
        s'.fileStore = (univ-sp) <: s.fileStore
}
```

that is,

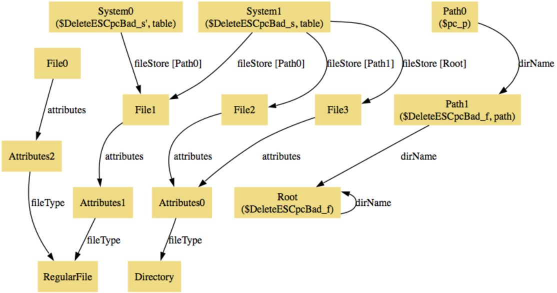
$$delete\ sp\ (M, N) \quad \triangleq \quad (M, N \cdot \Phi_{(\notin sp)}) \qquad (5)$$

where $\Phi_{(\notin sp)}$ is the coreflexive associated to the complement of sp.

Intuitive steps

Intuitively, *delete* will put at risk

- the *ri* constraint once we decide to delete file system objects which are open
- the *pc* constraint once we decide to delete directories with children.



(Model-checking in **Alloy** will easily spot these flaws, as checked above by a counter-example for the latter situation.)

Intuitive steps

We have to guess a **pre-conditions** for *delete*. However,

- How can we be sure that such (guessed) pre-condition is *good enough*?
- The best way is to calculate the weakest pre-condition for each constraint to be maintained.
- In doing this, mind the following properties of relational algebra:

$$h \cdot R \subseteq S \quad \Leftrightarrow \quad R \subseteq h^\circ \cdot S \tag{6}$$

$$R \cdot \Phi \quad = \quad R \cap \top \cdot \Phi \tag{7}$$

$$f \cdot R \subseteq \top \cdot S \quad \Leftrightarrow \quad R \subseteq \top \cdot S \tag{8}$$

For improved readability, we introduce abbreviations $ft := fileType \cdot attributes$ and $d := \underline{Directory}$, and **calculate**:

Calculational steps

$$
\begin{array}{ll}
 & pc(delete\ S\ (M, N)) \\
\Leftrightarrow & \qquad \{ \text{ (5) and (4) } \} \\
 & d \cdot (N \cdot \Phi_{(\notin S)}) \subseteq ft \cdot (N \cdot \Phi_{(\notin S)}) \cdot dirName \\
\Leftrightarrow & \qquad \{ \text{ shunting (6) } \} \\
 & d \cdot N \cdot \Phi_{(\notin S)} \cdot dirName^{\circ} \subseteq ft \cdot N \cdot \Phi_{(\notin S)} \\
\Leftrightarrow & \qquad \{ \text{ (7) } \} \\
 & d \cdot N \cdot \Phi_{(\notin S)} \cdot dirName^{\circ} \subseteq ft \cdot N \cap \top \cdot \Phi_{(\notin S)} \\
\Leftrightarrow & \qquad \{ \text{ } \cap\text{-universal ; shunting } \}
\end{array}
$$

Ensuring paths closure

$$\begin{cases} d \cdot N \cdot \Phi_{(\notin S)} \subseteq ft \cdot N \cdot dirName \\ d \cdot N \cdot \Phi_{(\notin S)} \subseteq \top \cdot \Phi_{(\notin S)} \cdot dirName \end{cases}$$

$\Leftrightarrow$ { $\top$ absorbs d (8) }

$$\begin{cases} \underbrace{d \cdot N \cdot \Phi_{(\notin S)} \subseteq ft \cdot N \cdot dirName}_{\text{weaker than } pc(N)} \\ \underbrace{N \cdot \Phi_{(\notin S)} \subseteq \top \cdot \Phi_{(\notin S)} \cdot dirName}_{wp} \end{cases}$$

Back to points, wp is:

$$\langle \forall\ q\ :\ q \in dom\ N \wedge q \notin S :\ dirName\ q \notin S \rangle$$

$\Leftrightarrow$ { predicate logic }

$$\langle \forall\ q\ :\ q \in dom\ N \wedge (dirName\ q) \in S :\ q \in S \rangle$$

Ensuring paths closure

In words:

if parent directory of existing path q is marked for deletion than so must be q.

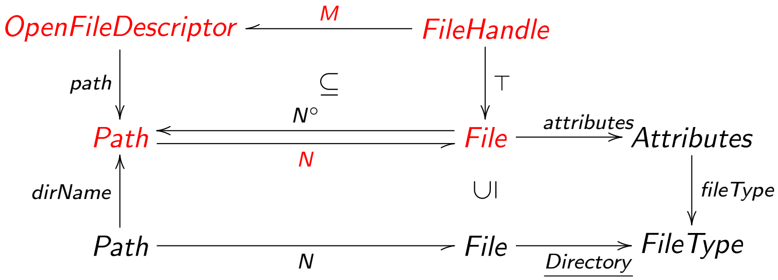
Translating calculated weakest precondition back to Alloy:

```
pred pre_delete[s: System, sp: set Path]{
     all q: Path |
        some q.(s.fileStore) &&
             q.dirName in sp => q in sp
}
```

Back to the diagram

PF-encoding of model constraints in terms of relational composition has at least the following advantages:

- it makes **calculations** easier (rich algebra of $R \cdot S$)
- it makes it possible to **draw** constraints as rectangles in diagrams, recall



- it enables the "navigation-styled" notation of Alloy

Constraint bestiary

- Experience in formal modeling tells that designs are **repetitive** in the sense that they instantiate (**generic**) constraints whose ubiquitous nature calls for classification
- Such "**constraint patterns**" are rectangles, thus easy to draw and recall
- In the next slides we browse a little "constraint bestiary" capturing some typical samples.

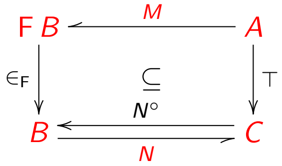
Constraints are Rectangles

- All of shape

$$R \cdot I \quad \subseteq \quad O \cdot R$$

- Example: **referential integrity** in general, where N is the *offer* and M is the *demand* :

$$\rho\,(\in_{\mathsf{F}} \cdot M) \subseteq \delta\, N \quad \Leftrightarrow$$



$$\Leftrightarrow \quad \in_{\mathsf{F}} \cdot M \subseteq N^{\circ} \cdot \top$$

M, N simple. $\in_{\mathsf{F}}$ is a membership relation.

Constraints are Rectangles

- **Example:** M, N domain-disjoint

$$M \cdot N^{\circ} \quad \subseteq \quad \bot$$

- **Example:** simple M, N domain-coherent

$$M \cdot N^{\circ} \quad \subseteq \quad id$$

- **Example:** M domain-closed by R:

$$M \cdot R^{\circ} \subseteq \top \cdot M$$

(path-closure constraint pc instance of this)

- **Example:** range of R in Φ

$$R \subseteq \Phi \cdot R$$

Last but not least

- Rectangles enforce **composition** as main relational operator (as in **Alloy**) which is the **multiplicative** operator in the abstract notion of a computation captured by

  Semirings $(S, +, \cdot, 0, 1)$ *inhabited by computations (eg. instructions, statements) where*

  - $x \cdot y$ *captures* **sequencing**
  - $x + y$ *captures* **choice** *(alternation)*
  - 0 *means* **death**
  - 1 *means* **skip** *(do nothing)*

- Theorem provers such as **Prover9** are especially apt to deal with this kind of structures [3].
- Thus "rectangular" Alloy indeed offers the *other side of the moon* — an effective connection to theorem proving

What will keep us busy for a while

Current work:

- Defining a simple pointfree **binary** relational semantics for **Alloy**, hopefully simpler than that of [2] (see appendix)
- (Future: base the conversion of pointwise to PF Alloy on such semantics)
- Studying the translation to/from Haskell and, in particular, how to port counterexamples to QuickCheck.

Near future:

- Connect **Alloy** to **Prover9** and **Mace4**
- Start the design of an Alloy-centric **tool-chain** incorporating new tools (the MEDEA project)

Appendix — semantics of "dot join"

Meaning of $R.S$ in Alloy:

$$[\![R.S]\!] = [\![S]\!] \cdot [\![R]\!] \qquad A \xrightarrow{[\![R]\!]} B \xrightarrow{[\![S]\!]} C$$

wherever both R, S are binary relations, or

$$[\![s.S]\!] = \underbrace{\rho\,([\![S]\!] \cdot [\![s]\!])}_{sp\ S\ s} \qquad B \xrightarrow{[\![s]\!]} B \xrightarrow{[\![S]\!]} C \xrightarrow{[\![s.S]\!]} C$$

wherever s is a set (unary relation) and R is binary. (Read $sp\ R\ s$ as meaning the *strongest post-condition* ensured by R once pre-conditioned by s.)

Appendix — semantics of "dot join"

Since $s.\tilde{~}r$ is equal to $r.s$ (as postulated in the Alloy book [4]), that is

$$[\![S.s]\!] \;=\; [\![s.\tilde{~}S]\!]$$

holds, then

$$[\![S.s]\!] = \underbrace{\delta\,([\![s]\!] \cdot [\![S]\!])}_{wp\ S\ s}$$

(Read $wp\ S\ s$ as meaning the *weakest pre-condition* required for S to ensure s on its output.)

In case R is a function f and s is a scalar x, than $[\![x.f]\!]$ simply means function application $f(x)$.

Example — dot join associativity

Let us check under what conditions equality

$$(x.r).s \quad = \quad x.(r.s) \tag{9}$$

holds in Alloy.

In case of binary relations we are done.

The case where x is unary (a set) follows in the next slide, where uppercase letters denote binary relations.

Example — dot join associativity

$$
\begin{aligned}
& [\![(x.R).S]\!] = \rho\,([\![S]\!] \cdot [\![(x.R)]\!]) \\
\Leftrightarrow \quad & \{ \text{ definition } \} \\
& [\![(x.R).S]\!] = \rho\,([\![S]\!] \cdot \rho\,([\![R]\!] \cdot [\![x]\!])) \\
\Leftrightarrow \quad & \{ \text{ range of composition } \} \\
& [\![(x.R).S]\!] = \rho\,([\![S]\!] \cdot ([\![R]\!] \cdot [\![x]\!])) \\
\Leftrightarrow \quad & \{ \text{ standard binary relation associativity } \} \\
& [\![(x.R).S]\!] = \rho\,(([\![S]\!] \cdot [\![R]\!]) \cdot [\![x]\!]) \\
\Leftrightarrow \quad & \{ \text{ } R \text{ and } S \text{ are binary } \} \\
& [\![(x.R).S]\!] = [\![x.(R.S)]\!]
\end{aligned}
$$

Example — dot join associativity

Wherever only the middle component is unary, associativity

$$(R.x).S = R.(x.S)$$

holds under side condition $\tilde{}R.S = S.\tilde{}R$.

It never holds in case of multiary relations — the equality doesn't even type check!

The general rule, as in Alloy's book [4]:

> *If two ways to parenthesize a join expression are both well formed they will be equivalent.*

What does "well formed" mean? Currently formally checking **informal** statements such as this in the book.

P.J. Freyd and A. Ščedrov.
Categories, Allegories, volume 39 of *Mathematical Library*.
North-Holland, 1990.

Marcelo F. Frias, Carlos G. Lopez Pombo, Gabriel A. Baum, Nazareno M. Aguirre, and Thomas S.E. Maibaum.
Reasoning about static and dynamic properties in alloy: A purely relational approach.
ACM Trans. Softw. Eng. Methodol., 14(4):478–526, 2005.

Peter Höfner and Georg Struth.
On automating the calculus of relations.
In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *IJCAR*, volume 5195 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 2008.

D. Jackson.
Software abstractions: logic, language, and analysis.
The MIT Press, Cambridge Mass., 2006.
ISBN 0-262-10114-9.

Joost Visser.
Real estate exchange.
Technical report, DI/UM , Braga, Jan 2007.
PortoDigital – SEC-11. Confidential.

P.L. Wadler.
Theorems for free!
In *4th International Symposium on Functional Programming Languages and Computer Architecture*, pages 347–359, London, Sep. 1989. ACM.